MIKELANGELO

D2.20
The intermediate MIKELANGELO architecture

Workpackage	2	Use Case & Architecture Analysis	
Author(s)	Nico Struckmann		USTUTT
	Carlos Díaz		USTUTT
	Uwe Schilling		USTUTT
	Peter Chronz		GWDG
	Nadav Har'El		SCYLLA
	Gregor Berginc		XLAB
	Shiqing Fan		Huawei
	Yossi Kuperman		IBM
	Joel Nider		IBM
Reviewer	John Kennedy		INTEL
Reviewer	Joel Nider		IBM
Dissemination Level	Public		

Date	Author	Comments	Version	Status
22 April 2016	Nico Struckmann	Initial structure.	V0.0	Draft
15 June 2016	All authors	Initial content provided for all sections	V0.1	Draft
20 June 2016	All authors	Refined content, addition of	V1.0	Review

		requirements and outlooks.		
30 June 2016	Nico Struckmann, Gregor Berginc, Uwe Schilling	Document ready for submission	V2.0	Final

Executive Summary

This report describes the current state of the overall MIKELANGELO architecture and provides outlook for the next phase in the development of the MIKELANGELO technology stack. The report starts with presentation of the two main layers MIKELANGELO seeks to optimise, namely the hypervisor and the virtual operating system. Each details the progress to date and discusses remaining requirements that have been gathered during the design and implementation phases.

The report continues with the introduction of the two overarching architectures that the project is interested in: cloud and HPC-Cloud. The merger of the two architectures and approaches to management of compute and storage resources seems to be inevitable. However, even the current state of the art virtualisation solutions do not yet allow for I/O performance near the one guaranteed by the hardware components. It is estimated that big data applications, one of the main users of HPC infrastructures in the future, will manage geopbytes $(10^{30}\ bytes)$of data by 2020. Exascale means that applications will run at 10^{18}FLOPs (floating-points operations per second) with a node concurrency of thousand nodes per application sharing data between them. These requirements make the efficiency in I/O operations at each level of the infrastructure architecture, chip-to-memory, memory-to-node and node-to-storage, the key factor to achieve the highest possible performance[1].

The advances in individual components we have made based on the initial architecture design already demonstrate performance improvements for I/O operations in virtualised environments. Preliminary evaluations using synthetic workloads and our business use cases have also revealed limitations as well as new possibilities for additional optimisations. These are also presented in this report.

MIKELANGELO project realised that providing only performance improvements does not guarantee the exploitation potential of our work. To this end, significant effort focuses on the improved application management and delivery system, an overarching, fully customisable telemetry system, and enhanced security monitoring and mitigation.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

[1] Joint Workshop on Large-Scale Computer Simulation. March 9-11, 2011, Aachen/Jülich (Germany), http://www.grs-sim.de/cms/upload/Workshop/dongarra_lss.pdf

Table of contents

1 Introduction 9
2 The Intermediate sKVM Hypervisor Architecture 11
2.1 Introduction and Previous Work 11
2.2 IOcm 13
2.2.1 IOcm Future Work 14
2.3 Virtual RDMA 15
2.3.1 Virtual RDMA Future Work 18
2.4 SCAM 20
2.4.1 SCAM Future Work 22
2.5 Key Takeaways / Concluding Remarks 22
3 Guest Operating System Architecture 24
3.1 Introduction 24
3.2 OSv 26
3.2.1 New Requirements for OSv 28
3.3 Seastar 29
3.3.1 New Requirements for Seastar 32
3.4 Application Management Tool 34
3.5 Application Packages 37
4 The Intermediate MIKELANGELO Architecture 40
4.1 Goals 40
4.2 Cross-layer Optimization 41
4.3 Integrated Cloud Infrastructure 43
4.3.1 IOcm 43
4.3.2 Virtual RDMA 44
4.3.3 SCAM 44
4.3.4 OSv 45
4.3.5 Snap 46
4.3.6 Application Package Management 48
4.4 Integrated HPC Infrastructure 49

4.4.1 Torque Extensions for Virtual Machines in General 49
4.4.1.1 Scripts for VM-based Job Execution 50
4.4.1.2 Misc Files 52
4.4.1.3 Components Integrated 52
4.4.1.4 VM Parameter Extensions for Job Submission 54
4.4.1.5 VM Parameter parsing 54
4.4.1.6 Global and Per User Configuration 55
4.4.1.7 Environment Variables 55
4.4.1.8 Workflow / Sequence Diagrams 55
4.4.1.9 Considerations for Applications / Images 63
4.4.1.10 Limitations 63
4.4.2 Torque Extensions for Standard Linux Guests 64
4.4.2.1 System Image Requirements 64
4.4.2.2 Customization of Virtual Standard Guests 64
4.4.2.3 Preparation of Virtual Job Environment 66
4.4.2.4 Script Execution 66
4.4.2.5 Virtual Node Access 66
4.4.3 Torque Extensions for OSv 66
4.4.3.1 Image Requirements 67
4.4.3.2 Customization of Virtual OSv Guests 67
4.4.3.3 Preparation of Virtual Job Environment 67
4.4.3.4 Script Execution 68
4.4.3.5 Virtual Node Access 68
4.4.3.6 Execution of Integrated Components 68
4.4.4 IOcm 68
4.4.5 Virtual RDMA 69
4.4.6 Snap 71
4.4.7 SCAM and Torque NUMA Nodes 72
4.4.8 OSv Support for HPC Workloads 72
4.5 Requirements and Future work 75

4.5.1 Cloud 75
4.5.2 HPC 76
4.6 Key Takeaways / Concluding Remarks 77
5 Concluding Remarks 78
6 References and Applicable Documents 79
Appendix A. Technical Details on the HPC-Cloud Infrastructure 81
A.1 Virtual RDMA Set-Up and Tear-Down Commands 81
A.2 Snap Metrics 83
A.3 The qsub VM Parameters 84
A.4 Scripts for VM-based Job Execution 86
A.4.1. Common Files 86
A.4.2. Job submission wrapper cmd line tool 86
A.4.3. Node Scripts 87
A.4.4. VM Management Scripts 87
A.4.5. Root VM Scripts 88
A.4.6. Job Execution Wrapper Script 88
A.5 Misc Files 89
A.5.1. VM related files 89
A.5.2. Host OS related files 89
A.6 Configurable Parameters 89
A.7 Default Values for VM-based Jobs 91
A.8 Environment Variables 92
A.8.1. Global Environment Variables 92
A.8.2. User Environment Variables 92

Table of Figures

Figure 1. High-level Cloud and HPC-Cloud architecture. 9
Figure 2. sKVM Architecture. 12
Figure 3. Architecture overview of the Virtual RDMA design prototypes. 16
Figure 4. Lightweight RDMA Virtualization - Design Prototype II. 19
Figure 5. Initial architecture of the MIKELANGELO Package Management. Components marked with green color represent the main development focus. 35
Figure 6. scripts, templates and configuration files. 51
Figure 7. Misc files for VM based job execution. 52
Figure 8. Components integrated into virtualised HPC. 53
Figure 9. Extended Job Submission Sequence. 56
Figure 10. Extended Prologue Sequence. 57
Figure 11. Guest's Boot Sequence. 59
Figure 12. Job Execution Sequence. 60
Figure 13. Extended Epilogue Sequence. 61
Figure 14. Guest's shutdown sequence. 62
Figure 15. Visualization of snap metrics with grafana [39]. 72

Table of Tables

Table 1. Verb calls that will be supported in prototype II. 18
Table 2. Specific cloud-init commands used for customization of virtual guests 65
Table 3. Several important environment variables that are configured by the startup script ... 81

1 Introduction

This document describes the current status and future work of the MIKELANGELO [1] architecture. The reports details all of its core components, ranging from an improved hypervisor and lightweight cloud operating system to cross-layer security and telemetry components. An approach to integration into high-level architectures is presented, based on two exemplary environments: HPC workload manager (Torque) and cloud middleware (OpenStack). This report accompanies the first public release of the MIKELANGELO consortium, so it offers some of the more detailed insights of the underlying components.

The MIKELANGELO architecture is designed to improve the I/O performance in cloud environments and also to bring the benefits of virtualization to HPC systems. These benefits include application packaging and deployment, and elasticity during the application execution, without losing the high performance in computation and communication of HPC systems. Although its design may look simple at first sight, the MIKELANGELO architecture is in fact a complex architecture as integrating all its individual components requires a large effort due to the number of interrelated technical challenges that need to be addressed.

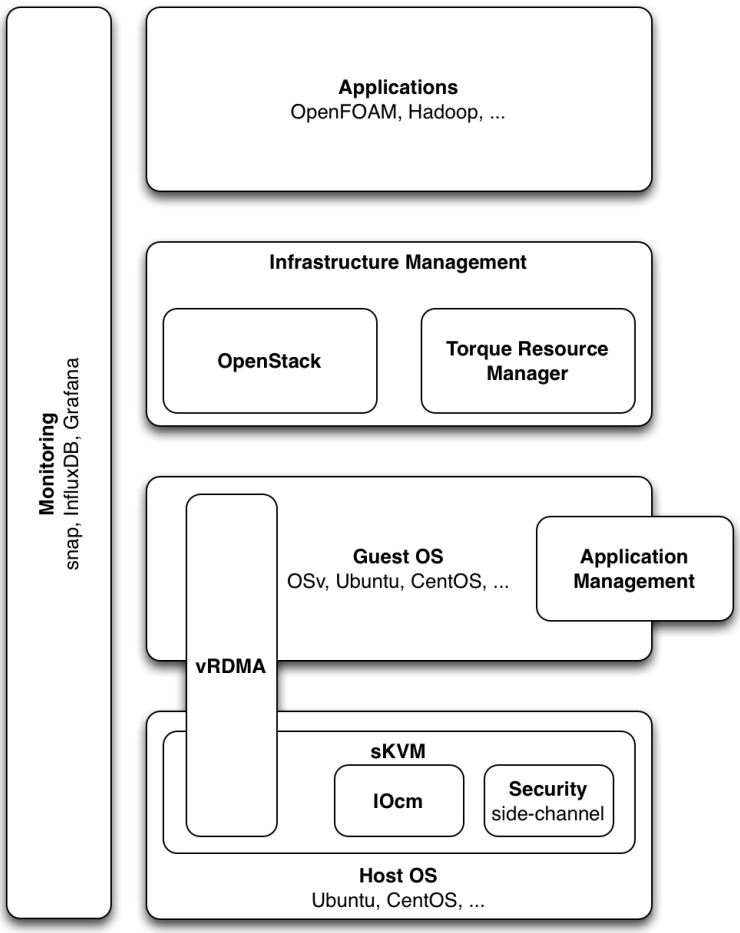

Figure 1. High-level Cloud and HPC-Cloud architecture.

This document is structured into the following chapters:

Chapter 2: The Hypervisor Architecture, sKVM. This chapter describes the work on improving performance and security at the lower layer of the architecture, the hypervisor. The work is split into three separate components: IOcm, an optimization for virtual I/O devices that uses dedicated I/O cores, virtual RDMA for low overhead communication between virtual machines, and SCAM, a security feature preventing side-channel attacks from malicious co-located virtual machines.

Chapter 3: The Guest Operating System Architecture, OSv. Chapter begins by introducing OSv, a guest operating system whose architecture has been already described in detail in deliverable D2.16 [2]. It continues with the description of efforts that were made to allow OSv to run Open MPI, a particularly important class of applications for HPC workloads. New requirements for the unikernel are presented next. Finally, a summary of work done and upcoming requirements for the Seastar framework is provided. Seastar is a novel C++ API and framework for efficient asynchronous server applications, based on several thorough analyses of popular cloud applications.

Another section is for the MIKELANGELO Package Management (MPM) architecture described in detail in deliverable D2.16 [2]. Advances made and new requirements in the different layers of the MPM architecture are reported here. The work on application management has allowed the consortium to provide a set of pre-built application packages. Commonly used packages, as well as some use case specific packages are presented at the end of this chapter as well.

Chapter 4: The MIKELANGELO Architecture is led by USTUTT and GWDG partners. It presents the two different perspectives on the potential integration of underlying components presented in previous chapters. Additional cross-layer optimizations are also discussed to support virtual environments, one of the main goals of the MIKELANGELO framework.

Each chapter consists of high-level description of the work done and remaining requirements for various components of the architecture. These requirements are either internal or external. Internal requirements are those that are identified by the component owner and are part of the initial design (for example, implementation of OSv image composition). External requirements are related to either use case or integration.

The report is finished with a section summarising the purpose of architecture design, current status, and future plans for the MIKELANGELO architecture.

Finally, in Appendix A we have provided relevant details on the HPC integration and usage. This appendix is targeting potential first adopters interested in a more detailed information of the components and relations between them.

2 The Intermediate sKVM Hypervisor Architecture

In the previous iteration of the architecture document, we described our initial vision of the sKVM hypervisor, which included 3 main features:

1) IOcm - an optimization for virtio-based virtual I/O devices using multiple dedicated I/O processing cores
2) Virtual RDMA - a new type of virtio device implementing the RDMA protocol for low overhead communication between virtual machines
3) SCAM - a protection mechanism to thwart side-channel attacks (such as cache sniffing) from malicious co-located virtual machines

These 3 features were implemented, measured and demonstrated during the first 12 months of the project. After the first iteration, we analyzed the work and have planned new research and development activities to advance each of these features. We have learned what works well and what doesn't, which assumptions and intuitions proved correct, and what has turned out to be flawed. Overall our progress has been positive, and our results are inline with our expectations. We now are pursuing the second phase of the research, which is to refine our designs and implementations based on what we have learned so far.

This section of the document briefly describes each feature with a summary of the work done so far, and then continues to describe the changes we are making in the second year.

2.1 Introduction and Previous Work

During the last reporting period the improved MIKELANGELO hypervisor sKVM emerged towards the full implementation of an evolved architecture that covered many requirements from the initial gathering phase, as well as new ones that came up during the integration into the Cloud and HPC infrastructures.

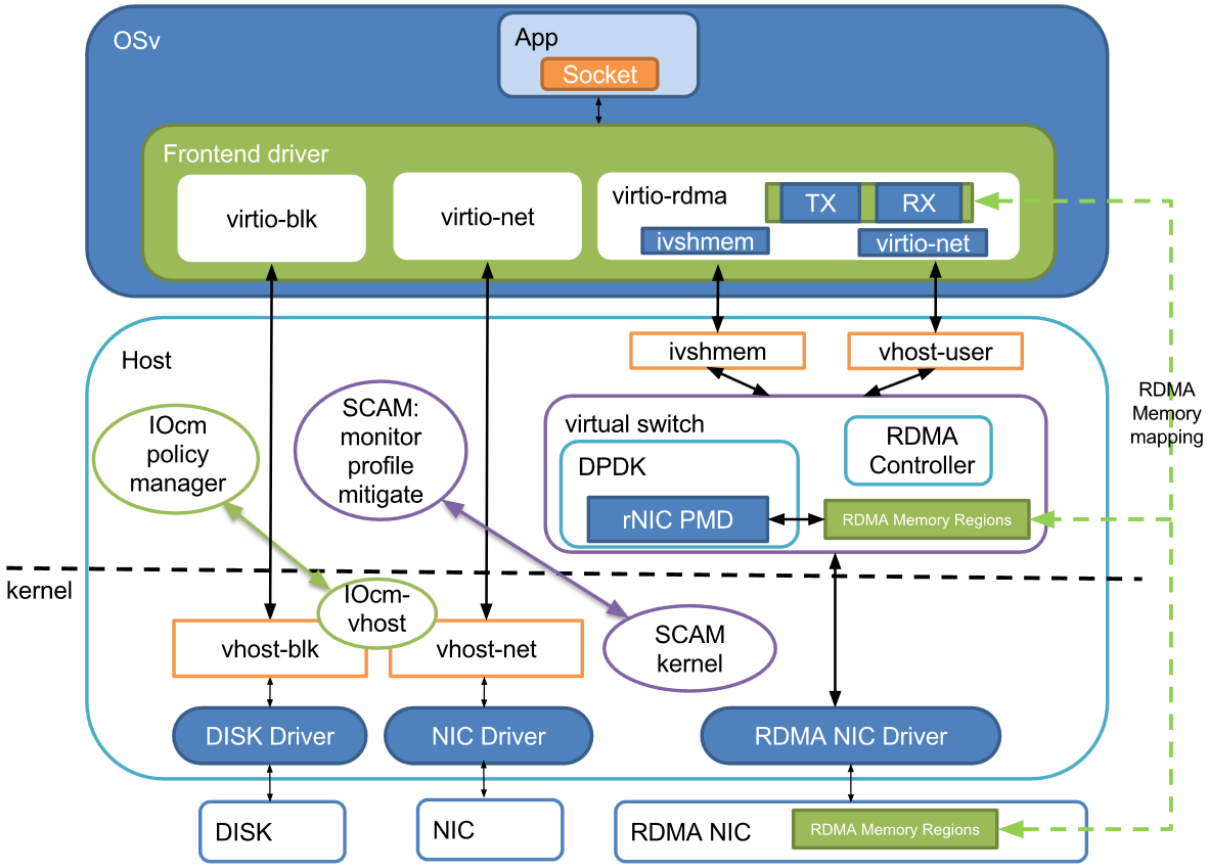

Figure 2. sKVM Architecture.

The sKVM architecture diagram shows the relationship of the various components inside the hypervisor, and interfaces to external components. There are three key architectural contributions from MIKELANGELO:

1. IOcm policy manager is user-space code which communicates with the IOcm vhost implemented in kernel-space.
2. Virtual RDMA consists of a virtio-rdma component in the virtual machine which communicates with the packet switch and NIC through shared memory.
3. SCAM monitor resides in user-space, and communicates with the SCAM kernel module.

The following subsections provide details about each of these individual components. Each subsection details the architectural changes we are implementing during the second year of the project. These are iterative changes based on the previous work (detailed in the previous section) which are influenced by the lessons learned during the evaluation of the first iteration. The main goal of the first iteration was to show a proof of the concepts at a functional level. The main goal of the second iteration is to show a performance improvement by modifying and optimizing the implementations of the first iteration. In some

cases this will require integration with the guest, and some interface changes that were not present in the first iteration.

2.2 IOcm

In the previous iteration of the architecture (D2.13, The first sKVM hypervisor architecture [3]), we defined the problem domain as being paravirtual I/O device inefficiencies in KVM, specifically as implemented in vhost, which has the highest throughput and lowest latency. IOcm incorporates and improves on IBM's background technology known as ELVIS, which in turns improves on the base vhost implementation by dramatically reducing latency in guest-to-hypervisor communication inside the paravirtual device. This reduction in latency is achieved by polling a set of virtual devices (belonging to one or more virtual machines) for I/O activity, and acting directly upon the requests as they are entered into the queue, rather than waiting for notification of the request. This polling mechanism works best when a physical core is dedicated to the task of I/O processing. However, dedicating a physical core to I/O processing introduces inefficiencies in the face of a changing workload. We find that a single dedicated core rarely provides the precise processing power required by the virtual machines, and is often either too much or too little for the job. IOcm improves on ELVIS by managing the CPU resources more effectively, by dynamically reassigning cores to I/O processing or to general purpose tasks as dictated by the current state of the workload. The dynamic reassignment of cores ensures the most effective resource allocation, allowing KVM to provide maximum throughput with reduced average latency per I/O request. IOcm takes control of the CPU core affinity settings of the VMs, and modifies the affinity when adding or removing IO cores. When running an I/O-intensive phase of an application, I/O processing time becomes the bottleneck, and additional CPU time is required in order to speed up the application. During such a phase, it is beneficial for IOcm to break any affinity for the virtual CPUs, to maximize I/O throughput. When the application starts a CPU-intensive phase, IOcm will relinquish control of the I/O cores, and reconfigure them to available for CPU processing. We rely on the host scheduler to make the correct decisions when scheduling the virtual CPU threads to get the best performance. The virtual CPU threads can have affinity for a group of CPU cores, but it does not make sense to a keep a 1:1 mapping, since there is no guarantee that such a mapping is possible (there may be more or fewer vCPU threads than available physical cores).

In the current implementation, the IOcm feature is composed of two sub-components. The first sub-component is in kernel space, and is implemented as a set of patches on top of the Linux kernel. The first version was implemented on top of kernel 3.9, but has been ported to Linux kernel version 3.18, as agreed upon by the consortium as our working version of the kernel. The second sub-component is a control application that resides in user space, and is used to monitor and manage the in-kernel sub-component through the sysfs interface

described in [4]. The implementation of the control program is only at preliminary stage, and is used to exercise the interface. It behaves autonomously (i.e. does not require administrator interference for correct operation) but does not yet contain the correct algorithm needed to achieve the maximum possible performance improvement from the shared vhost architecture.

The sysfs interface was chosen as an intermediate step, and is not intended to be used in the final version. We chose sysfs because it is a commonly used and accepted interface between kernel space and user space in Linux. It is very flexible, and allows us to implement all control and monitoring channels with a minimal amount of effort. The main drawback of using sysfs is that once it is published, it must be maintained. Any interface that the kernel exposes can potentially be used by any user space component, by any application in the world. If such an interface were to be removed or changed in a subsequent kernel release, it would break compatibility with any user space application relying on it, and thus destroy functionality. This is considered very bad practice in the Linux community (as well as in the larger software development community), and thus the kernel maintainers are very careful about including a new interface in released code. So while sysfs is an invaluable tool for development and debugging, its use in released code is marshalled very carefully to ensure "future-proofing". That means we will have to select an alternative method of controlling the vhost threads for the final version of the code.

As a result of our experimentation during the past year, we have discovered the limits of a shared vhost thread. We understand much better under which scenarios it provides a benefit, and how much of a benefit we can expect in such scenarios. We have also discovered additional sources of inefficiencies in the virtual I/O subsystem, and have started to consider the potential performance improvements we can expect from additional modifications.

2.2.1 IOcm Future Work

There are several aspects of IOcm that are being improved in year 2 of the project. Firstly, we plan to implement the complete algorithm in the control component to achieve maximum performance gains from the shared vhost thread. The algorithm works by comparing the loads on the two groups of cores (I/O cores vs. CPU cores). If the I/O cores are fully loaded, and the vCPU cores are not, there is an opportunity to reallocate one of the vCPU cores to boost I/O performance. Conversely, if the vCPU cores are fully loaded, but the I/O cores have a relatively low load, then it is possible that reallocating an I/O core as a vCPU core will give a performance boost. If both the I/O cores and the vCPU cores are fully utilized (this is the ideal, but rare case), it means the system is more or less balanced and no changes should be made. In all of these cases, there are many additional factors that must be taken into account in order to reach the final decision.

Secondly, we will investigate, choose and implement an alternative to the sysfs interface. One alternative is to move all control functions inside the kernel, thereby removing the need to maintain a kernel interface.

Lastly, we plan to investigate other possible improvements, based on our discoveries during the past year. One such avenue is to investigate the effect of caching on the virtual I/O messages. As messages are passed from virtual machines to the hypervisor (and vice versa) through shared memory buffers, this shared memory is often cached in L1 and L2 caches. The difference in access time between the L1 cache and main memory differs by machine and architecture, but is generally 2 orders of magnitude (100x). When these caches remain unperturbed during the life of the message, performance is considerably higher. However, due to a number of effects such as task switching and cache replacement, important data is often ejected from the cache too soon, leading to a performance hit. Another avenue to investigate is the possibility to remove the need to copy memory in the receive channel. When data is received by the hypervisor that is destined for a VM, it is copied several times as it passes through many layers of protocols before it reaches its final destination. When the payload is large, the time required for copying the data becomes the limiting factor in throughput measurements, and also adds to the latency. By removing the need to copy data, we can improve the throughput, and reduce the latency. However, this requires bypassing existing interfaces, which has an impact on the generality of the solution and may not be acceptable to the Linux community.

2.3 Virtual RDMA

In D2.13 [3] (The first sKVM hypervisor architecture), we introduced the background and challenges of RDMA virtualization. The main goal of the virtual RDMA is to provide better communication performance between VMs and high flexibility for the virtualization environment, by using a paravirtualized RDMA device, virtio-rdma, based on the virtio standard.

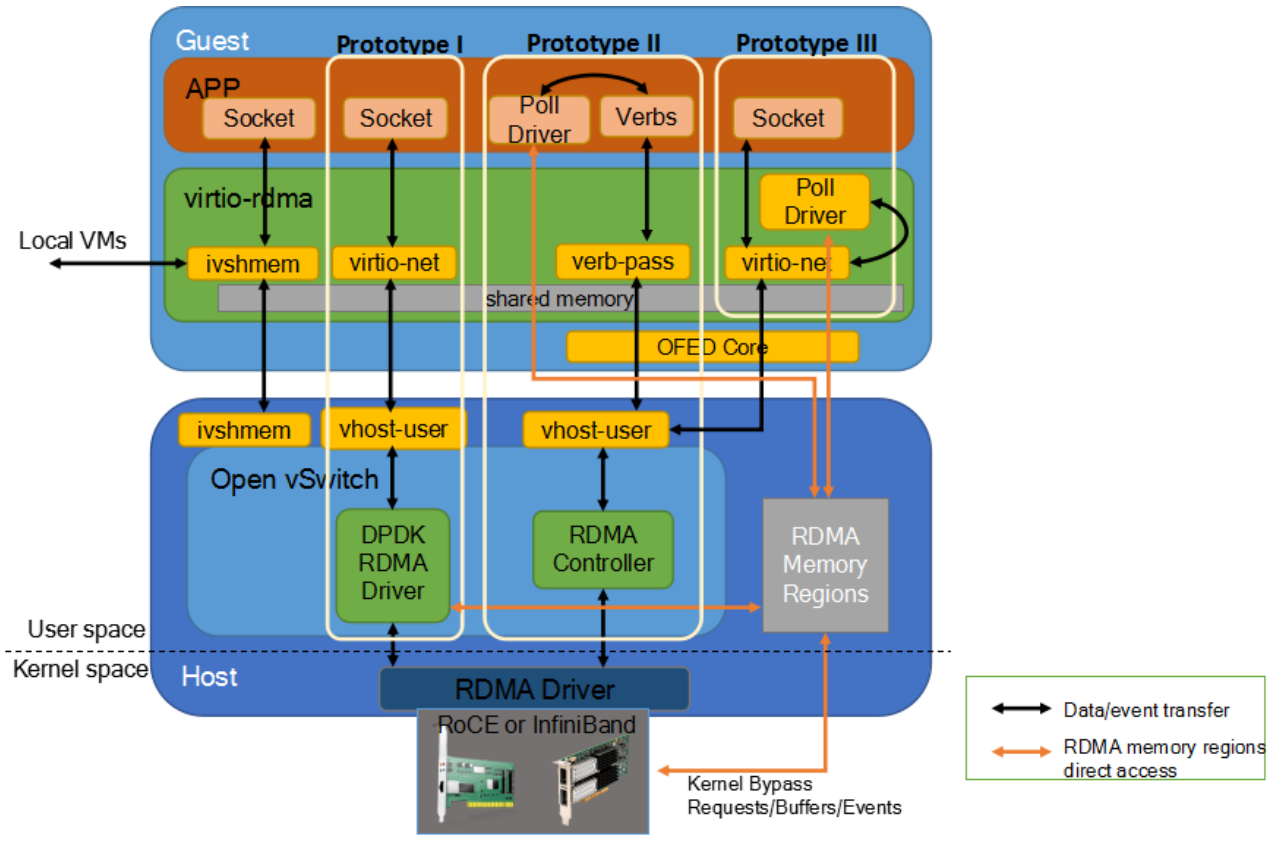

Figure 3. Architecture overview of the Virtual RDMA design prototypes.

Three solutions and design prototypes were proposed for different applicable scenarios that require no modification of the user application, as shown in Figure 3. Design prototype I is based on several open source packages, which can be directly used and integrated. It supports the sockets API for the guest applications. The network API calls are translated into RDMA verbs for the device by the DPDK rNIC PMD (Poll Mode Driver). The real communications are performed in the backend driver, and every single operation requires switching between the guest and the host many times, e.g. forwarding control commands and performing buffer operations. Design prototype II supports only the verbs API on the guest, and it is supposed to be the most efficient solution for communication between VMs. Communication buffers and completion events are managed and processed directly by the guest application. The involvement of the host is only for passing verbs to the RDMA device, and the performance is improved due to the kernel bypass feature of the RDMA implementation. Design prototype III keeps most of the advantages of prototype I and II. It supports the sockets API on the guest, but implements a PMD within each VM. The socket calls from the guest application are translated to RDMA verbs by virtio-rdma and passed to the backend driver, i.e. the RDMA controller. This is similar to the functionality of the DPDK rNIC PMD in design prototype I, but the translation tasks are now accomplished by the frontend driver. This will allow the frontend driver to actively poll the completion event and

avoid sending events between the guest and the host, which is expected to show a significant improvement in performance.

The implementation plan of prototype I, described in D2.13 [3], has been executed accordingly. In the guest, a virtio-net device is created to support the TCP stack. The backend driver using vhost-user on the host has been enabled by the combination of Open vSwitch, DPDK and QEMU. Open vSwitch provides a vhost-user-aware bridge, which connects the InfiniBand port to a DPDK vhost-user port. The Open vSwitch Daemon runs a PMD, shown as DPDK RDMA Driver in Figure 3, which controls the datapath between the physical port and the vhost-user port and converts the socket messages to the corresponding RDMA format. The created vhost-user port is then used by QEMU when starting the guest, and a link with the guest virtio-net device is created. The communication between guest and vhost-user port is controlled by QEMU through the shared memory implementation of vhost-user.

We have tested and compared the performance between several cases in D4.1 [5], which showed that prototype I has much better communication performance than the traditional and non-RDMA virtio-net implementation. Further performance tests on use cases are presented and evaluated in D6.1 "First report on the Architecture and Implementation Evaluation" [6]. Based on the performance tests, we have discovered the Open vSwitch Daemon, or, to be more precise, DPDK PMD, is the main performance bottleneck of prototype I, as each single communication needs to be processed by the PMD, which requires occupying full CPU cores and adds quite a big overhead in the communication path.

The implementation of ivshmem has been postponed due to its unexpected complexity and lower priority, and the current solution for shared memory communication is done with vhost-user directly. The performance of inter-VM communication across two hosts has been tested and benchmarked, as presented in D4.1 [5]. A more detailed performance test on an integrated HPC environment is discussed in D6.1 [6].

For HPC infrastructure, software packages are normally installed centrally on a shared network file system (NFS) to make it available to all compute nodes. The use of NFS introduces additional complexities for integrating the virtual RDMA prototype I into the HPC environment. For example, when all the compute nodes use the same binaries of Open vSwitch, DPDK, Libvirt and QEMU, it may result in access conflicts of the same context files, temporary data or any metadata that is stored in the centrally shared directory. We solved this problem by assigning different environment variables and isolating the metadata for different compute nodes in the HPC integration.

Detailed instructions for setting up and using virtual RDMA prototype I have been provided in D4.1's appendix [5] and Appendix A.1, including dedicated configuration instructions for

HPC environments. The example shell scripts provided for the HPC integration have been successfully integrated into USTUTT's HPC test environment.

The instantiation of the networking on the physical node for the virtual RDMA takes place in a job's root prologue script that runs first and is executed as user root. The cleanup, namely reverting the network configuration on the physical host back to its previous state, takes place at the end of a job's root epilogue, which is the last part in a batch job's life cycle.

Virtual guests are defined by a domain XML based on the user's resource requests and globally defined defaults. The virtual guests require some enhancements to its domain XML, as well as some additional binary packages installed, to utilize the virtual RDMA capabilities. Further, some environment variables are required to be set to define the location of required binaries.

2.3.1 *Virtual RDMA Future Work*

As described in D2.13 [3], virtual RDMA design prototype II (Figure 3) is aimed at supporting guest applications that directly use RDMA verbs, which are normally processed by the RDMA device driver. The pinned RDMA memory region is directly shared between the guest application and RDMA device. This allows the application to actively poll the Completion Queues for the best performance. However, in order to allow the guest application to directly access the memory regions, for example, to prepare the QPs and WRs, the guest application has to know the underlying RDMA device information. The host must expose part of the hardware information to the guest, which limits the migration of the virtualization system. This will require supporting and integrating part of the RDMA kernel drivers in the virtio-rdma virtual driver.

The basic verb calls [7] that are supported by the basic version of prototype II are shown in the following table. These verbs were collected using Valgrind [8] and Callgrind [9] on a parallel version of the OpenFOAM application for the Aerodynamics use case in D2.10 [10].

Table 1. Verb calls that will be supported in prototype II.

`ibv_get_device_list`	`ibv_open_device`	`ibv_create_cq`	`ibv_create`
`ibv_query_device`	`ibv_alloc_pd`	`ibv_close_device`	`ibv_create_qp`
`ibv_query_gid`	`ibv_reg_mr`	`ibv_get_deivce`	`ibv_destroy_srq`
`ibv_destroy_qp`	`ibv_destroy_cq`	`ibv_query_port`	`ibv_dealloc_pd`
`ibv_free_device_list`	`ibv_modify_srq`	`ibv_get_device_name`	`ibv_poll_cq`
`ibv_dereg_mr`	`ibv_post_srq_recv`	`ibv_post_recv`	`ibv_fork_init`

ibv_get_sysfs_path	ibv_post_send	ibv_get_async_event	ibv_ack_async_event

In the design of prototype II, as shown in Figure 4 below, the requests of using the above verb calls will be first processed by the frontend driver on the guest, and forwarded to the RDMA Controller (backend driver) on the host through the vhost-user connection. The controller will then perform the verb calls using the RDMA NIC driver and return the results back to the frontend driver and the application.

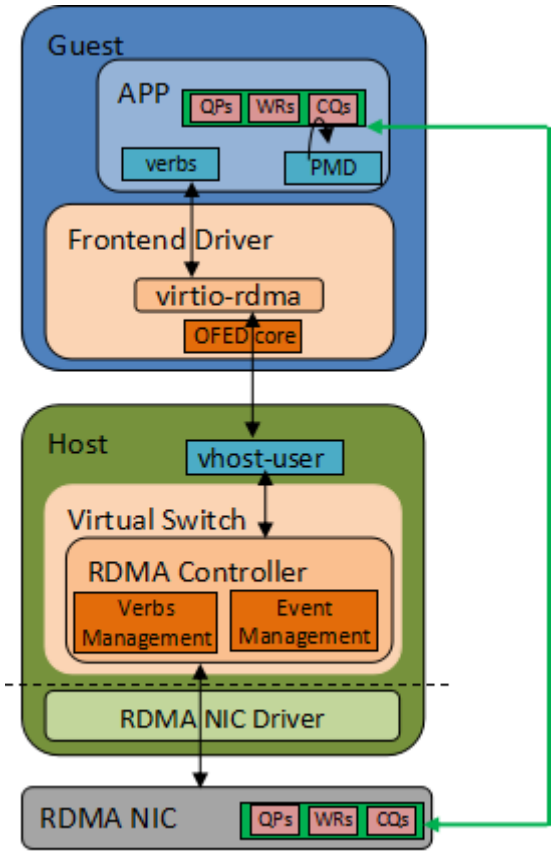

Figure 4. Lightweight RDMA Virtualization - Design Prototype II.

Research has been done on the available open source virtual RDMA implementations, and only one recently published design and implementation was found, namely Hyv [11, 12], which however is based on a rather old Linux kernel version (3.13 released in early 2014). It is a hybrid I/O virtualization framework for RDMA interface. The design of Hyv separates the hardware-dependent part from the hardware-independent part and eases the effort for vendors to implement their own virtual RDMA NICs. However, it is not a complete implementation that would support socket or MPI applications using verb calls, due to the missing support of various virtualized kernel modules, e.g. IPoIB (IP over InfiniBand), ib_mad, etc.

A deeper investigation of Hyv has been performed, in order to understand how much we can reuse the implementation. The Hyv implementation is very basic and limited. It doesn't simulate a general virtual device on the guest. Instead the user has to manually select one, which is based on a specific device version. This limits the extensibility of supporting different hardware on the host, and introduces extra configuration effort on the guest. For configuring Hyv, a few RDMA kernel modules have to be replaced by the instrumented ones, which proves that Hyv doesn't support all the RDMA functionality at moment. For example, IPoIB is not supported on the guest, because the ib_ipoib module has not been instrumented in Hyv. Another disadvantage of Hyv is that, whenever we want to use an additional InfiniBand kernel module in the guest, then we have to instrument a virtio version of it. This new instrumented kernel module will have to communicate to the corresponding new backend on the host. At the moment, there are only two Hyv virtual devices capable of communicating with two backend drivers on the host.

Although Hyv is different to what we have designed in prototype II, it can still be reused as the foundation of the implementation of prototype II.

We tested Hyv on the old Linux kernel it required in order to verify it's working correctly as is. Then the Hyv virtual drivers, as well as its QEMU patch, were ported to the targeted kernel version (3.18) for the MIKELANGELO project.

The further short-term plan is to resolve the bugs found in Hyv on the new Linux kernel, and then extend the Hyv frontend driver to a general virtio driver, i.e. virtio-rdma. This driver is finally going to take care of all the communications with the backend driver on the host. Similarly, we then need only one backend driver that performs the verbs calls. The longer plan until the end of the second project year, is to implement and release an initial version of prototype II, which should work with benchmarks and use cases in this project. The implementation will be integrated to the HPC and Big Data infrastructures, and the performance will be evaluated.

On the other hand, prototype I will continue to be supported and updated. Although the implementation works already with Ubuntu and OSv guests, there are still issues that should be solved to improve the usability and performance. This includes cooperating with Scylla to find a solution for the NUMA limit of OSv, and cooperating with GWDG to integrate the implementation on the Big Data infrastructure that is similar to the HPC integration at USTUTT.

2.4 SCAM

In D2.13 [3] (The first sKVM hypervisor architecture) we introduced the threat of cache side-channel attacks, along with an overview of potential methods for monitoring, profiling, and

mitigating such attacks. The main goal of the SCAM (Side Channel Attack Monitoring and Mitigation) module within sKVM is to minimize this attack surface which is a key source for concern in multi-tenant virtualized systems. As part of our preparatory work for SCAM, we have implemented a full prime-and-probe L3 cache side-channel attack that serves as a benchmark for testing and evaluating the SCAM module.

SCAM is designed to collect data on the pattern of cache access of virtual machines running over (s)KVM and use this data to profile VMs and evaluate the likelihood that a monitored VM is executing a cache side-channel attack. In this phase of the implementation, this monitoring activity is based on monitoring a collection of system counters via PAPI (Performance Application Programming Interface) in user-space. The complete list of counters is described in D3.4 [13]. This choice of counters is motivated by our benchmark attack implementation, which is highly discernable by these counters. This is mostly observable during the phase where set-mapping is performed (see D3.4 [13] for details on this phase). However, one cannot assume that an attacker would necessarily be forced to perform this phase, since this phase is platform specific, and may well be done offline by the attacker, well before it actually initiates the attack on the machine shared by both the target and the attacker. We therefore perform much closer monitoring of L2-level cache counters, where we are able to identify the attacker's behavior even in the phase where the relevant sets are being primed-and-probed. The next phase in the implementation of this part of the module will further explore the ability of counters to reveal malicious behavior by the attacker. Alongside this counter-monitoring approach, we plan to further explore additional methods for identifying an attack is taking place via performing a prime-and-probe sequence by the SCAM module, and also exploring the potential of emulating the VM's activity in order to identify non-benevolent behavior.

In terms of mitigation, the current implementation of the SCAM module focuses on "*Noisification*" of the attacker process. In this mitigation method our primary goal is to disrupt the activity of a potential attacker that may, or may not, be co-located with a target VM. To this end we have implemented a *ghost* target VM that uses shared-memory with the actual target VM (similarly to having deduplication of memory resources among two VMs), where the ghost target uses a different private key than that one used by the actual target. By having this ghost VM constantly perform square-and-multiply operations alongside the target VM, we are able to introduce significant amounts of noise into the readings performed by the attacker, and by that significantly hinder its ability to extract the true target VM's private key. In the next phase in this method of mitigation we plan to explore the noise introduced by running a ghost attacker. We further plan to combine the outputs of our monitoring mechanism as a tool for focusing the noise generation in the specific set, or sets, being explored by the attacker.

2.4.1 SCAM Future Work

We foresee three main activities in the next few months to develop and improve SCAM. These are monitoring, attack mitigation via introduction of noise, and attack mitigation via physical page partitioning. All these activities will be supported by continual improvement of the attack in order to test our techniques.

We expect that the monitoring effort will need to proceed in one of two directions. The first is relatively large scale collection of data from the PAPI package and usage of machine-learning methods to discern whether an attack is taking place. The second is tighter interaction with the potential attacker including prime and probe or some type of partial emulation of its execution in order to determine its cache access pattern.

The injection of noise to the system is currently carried out by a ghost implementation of the target. If the monitoring sub-module becomes precise enough to identify distinct cache sets that are being monitored it may be possible to use this information to introduce noise more precisely.

The objective of the page partitioning effort is to manipulate the page table so that a portion of the L3 cache is dedicated only to sensitive memory pages of the potential target. This approach is promising in that if such a partition is achieved then cache attacks by a VM are no longer possible on the protected area. However, certain key questions must be answered before this technique can be used. One of them is the applicability of the technique when large pages are used, or conversely assessing the performance downgrade if large pages aren't used. The second is the way in which SCAM is made aware of the sensitive memory pages. One option is by cross-layer interaction with the application and another is by using an accurate (and sophisticated) monitoring module.

2.5 Key Takeaways / Concluding Remarks

We have shown our research directions for the 3 features of sKVM during the second year of the project. These directions are a continuation of what was proposed at the beginning of the project, but influenced by what we have learned during the first year. At the end of the second year, we expect to have the 3 features integrated into a single working version of sKVM, showing performance improvements in their respective areas.

We will continue to implement, evaluate, analyze and modify throughout the year in an iterative manner to fine-tune each of the features. As the sKVM hypervisor forms the basis of the MIKELANGELO architecture, it is important that we be able to deliver our new features on schedule, and with the agreed interface for easy integration with the guest OS. We believe

that the architecture changes described above show how this can be accomplished, generating publishable results that we can be proud of.

3 Guest Operating System Architecture

The guest operating system (or "guest OS") is the operating system running inside each individual VM (virtual machine) of the MIKELANGELO cloud. The guest OS implements the various APIs (Application Programming Interfaces) and ABIs (Application Binary Interfaces) which the applications use. The goals of developing the guest operating system within the MIKELANGELO project are to make it easier to run I/O-heavy and HPC (high performance computing) applications in the cloud, and additionally to run such applications more efficiently than on traditional operating systems.

We described the architecture of the MIKELANGELO guest OS in detail in deliverable D2.16 [2], "The First OSv Guest Operating System MIKELANGELO Architecture". We also described its first implementation in detail in the deliverable D4.4 [14], "OSv - Guest Operating System - First Version". So we begin this section with only a brief introduction to the overall architecture of the MIKELANGELO guest OS, and the reader is encouraged to refer to these two previous deliverables if more details are desired. Thus, the focus of this section, after the following brief introduction, is to survey the progress that the guest OS architecture has made in the last year, to describe what requirements drove this progress, and to describe the requirements that will drive the continued development in the current period.

3.1 Introduction

Today, most cloud and HPC applications are written to run on Linux, the same OS that was used earlier when running on physical machines. Some of the features that once made Linux desirable on physical machines, such as a convenient single-machine remote administration interface (ssh, config files, etc.) and support for a large selection of hardware, now became irrelevant or even a burden when running on virtual machines as they increase Linux's size, complexity, and boot time. Some of the traditional roles of the OS have become redundant in the cloud, and are now pure overhead: the most prominent example is Linux's support for running multiple processes isolated from one another, and all of them isolated from the kernel. The cloud already offers isolation between the different VMs, so increasingly users are deploying separate applications on separate VMs instead of separate processes on the same VM. This makes isolation inside a VM redundant, and a performance burden because it slows down context switches, system calls, and other parts of the kernel.

Thus, MIKELANGELO replaces the Linux kernel and its system libraries with **OSv**, a new operating system designed especially for running efficiently on virtual machines, and capable of running existing Linux applications with certain limitations (namely, that the application is multi-threaded but not multi-process, and that it is compiled as a relocatable executable). Compared to Linux, OSv has a smaller disk footprint, smaller memory footprint, faster boot

time (sub-second), fewer run-time overheads, faster networking, and simpler configuration management.

The idea of using a specialized operating system for running a single application on each virtual machine has amassed significant popularity in the past year, and the term *unikernel* has become popular for this idea. While OSv is no longer the only unikernel in the market, it is one of the most mature implementations, and supports a variety of hypervisors, clouds, and applications, so it makes a good choice for MIKELANGELO which aims to support several use cases.

OSv is an open-source project which was started prior to the MIKELANGELO project by one of the MIKELANGELO partners (ScyllaDB, formerly Cloudius Systems). While OSv was more mature than alternative unikernels, it still needed significant improvements to become useful for MIKELANGELO. The work in WP2 of analyzing OSv's architecture and the various use cases which we intend to run on it, resulted in a set of required improvements to OSv which are being advanced within WP4. This continued development of OSv for MIKELANGELO considers performance, usability and application compatibility, and in the next section we will survey the improvements to OSv made - and the MIKELANGELO requirements that led to them.

While OSv allows running existing Linux applications, we noted in D2.16 [2] that certain Linux APIs, including the socket API, and certain programming habits, make applications which use them inefficient on modern multi-core hardware. OSv can improve the performance of such applications to some degree, but rewriting the application to use new non-Linux APIs can bring significantly better performance. So in MIKELANGELO we also introduced a new API, called **Seastar**, for writing new highly-efficient asynchronous server applications. One of the four use cases we analyze and implement in WP2 and WP6, the "Cloud Bursting" use case, can particularly benefit from Seastar; This use case is based on the Cassandra distributed database, and reimplementing Cassandra with Seastar resulted in as much as 10-fold performance improvement over regular Cassandra. Seastar, and also the Cassandra reimplementation using Seastar (called "scylladb"), are also open source projects. The improvements we are making to Seastar as part of the MIKELANGELO project include improving Seastar's architecture and implementation to make it faster and more useful for MIKELANGELO's specific use case (Cloud Bursting), but also making Seastar more generally useful for more potential MIKELANGELO users by improving Seastar's design, features, and documentation.

Beyond the benefits of modifying just the guest OS, we can also benefit from possible synergy with the hypervisor, which MIKELANGELO also modifies (as described in the previous section): We can get additional performance benefits from modifying both layers in a

cooperative manner. The main cross-layer (guest OS and hypervisor) improvement we introduced in MIKELANGELO is **virtual RDMA** (virtualized Remote Direct Memory Access). We already discussed this cross-layer feature in the previous section of this document (the hypervisor), so we will not address it again in this section. Another aspect of the guest OS we will not address here is OSv's extensive monitoring capabilities via a REST API, which are mostly unchanged from their previous description in D2.16 [2].

In addition to improving efficiency, another goal of the MIKELANGELO project is to simplify deployment of applications in the cloud. MIKELANGELO introduces the MIKELANGELO Package Manager (**MPM**), a package management and image composition tool which allows users to quickly and conveniently compose VM images which are ready to run on the cloud. MPM allows a user to easily combine several existing application packages (libraries, components), together with the OSv kernel, into a stand-alone image ready to run on the cloud.

3.2 OSv

In D2.16 [2] we presented the initial architecture of OSv, the Linux-compatible unikernel which MIKELANGELO adopted as a base for its guest operating system. While at that stage OSv could already run numerous applications, it was not as mature as Linux and still missed a number of important features needed by MIKELANGELO's use cases. In WP2, our goal was to discover which such features were still missing, buggy, or perform unacceptably in OSv, and list them as requirements, which resulted in work items for WP4.

This year, we have focused much of our development and testing efforts around the Aerodynamic Maps use case, which required running the *Open MPI* HPC library and the *OpenFOAM* aerodynamic simulation application. This use case was particularly interesting to start with, because it interests several MIKELANGELO partners; More importantly, once we get this use case to work (and work well) on OSv, it will become easy to run other use cases based on Open MPI (such as the Cancellous Bones Simulation use case), and basically, most HPC workloads, since most of them are based on MPI.

While trying to run the MIKELANGELO use cases on OSv this year, we formulated the following requirements for OSv, which we later implemented in WP4 and contributed to the main OSv source code tree:

1. **Implement missing system calls and C library functions:**
   Although dozens of different Linux applications could already run on OSv, there still remained a tail of rarely used system calls and C library functions which OSv had not yet implemented, or implemented incorrectly due to lack of testing. Because Open MPI and OpenFOAM are large and complex packages (spanning several millions lines

of code), unsurprisingly they used several of these unimplemented or mis-implemented functions, which we discovered and listed as requirements for WP4. Some of the many examples include better support for `getrlimit()`, edge cases of `unlink()`, support for various system calls being called through `syscall()`, `__asprintf_chk()`, `memalign()`, `faccessat()`, `fstatat()`, `malloc_hooks`, `get_mempolicy()` and many more.

2. **Improve and debug OSv's build system:**
   While developing MIKELANGELO's package manager (MPM), see details below, we came across bugs and missing features in OSv's build system, which needed to be fixed.
3. **Support newer build environments:**
   OSv was developed using specific versions of the C++ compiler (gcc) and libraries (libstdc++, Boost, etc.), In MIKELANGELO and in general, different users often have different build environments, running different versions of the compiler, libraries, and other tools. Moreover, while we were working on MIKELANGELO, newer versions of these tools and libraries kept coming out. Some of these upgrades resulted in breakage of OSv - sometimes because of new bugs in these tools, but sometimes because of old bugs in OSv which surfaced with the new compiler. We had to fix or work around these bugs to get OSv to compile on a large variety of build environments. This is important for MIKELANGELO partners (who don't necessarily have identical build machines), but also for the general OSv users.
4. **Add NFS client to OSv:**
   OSv did not include support for an NFS client (i.e., mounting shared directories over the NFS protocol), but NFS is often used in HPC workloads. So we added this requirement for OSv, and implemented it as already described in D4.4 [14].
5. **Partial form of thread isolation in OSv:**
   As a matter of design principle, OSv does not support classic Unix "processes", i.e., threads which are fully isolated from each other. However, while running our Open MPI-based use cases we realized that in some cases, we do need some minimal level of isolation beyond that which threads can traditionally offer. In particular, two isolation requirements that arose were:
   a. We need to be able to run the same executable twice in two threads, without the two copies sharing their global variables, Open MPI needs this for running a separate copy of the parallel program on each core.
   b. We need to be able to give each of these threads different environment variables (in the sense of C's `getenv()`). Open MPI uses these environment variables to tell each thread its role in the computation.

3.2.1 *New Requirements for OSv*

Much of the improvements we did to OSv this year have been *reactive*, i.e., we tried to run various MIKELANGELO uses cases and partners' test cases and benchmarks on OSv, and discovered what we need to fix or improve. We made significant progress this year - previously OSv could not run Open MPI and OpenFOAM at all, and now it can, in various complex setups (multiple cores, multiple machines, etc.). We expect that a significant percentage of our work on OSv next year will also be reactive in this sense: We will continue to discover unexpected bugs, as well as small Linux features not yet supported by OSv, and need to solve these previously undetected problems.

Beyond that, there are already a few requirements which we discovered this year, but have not yet implemented. We will need to address these in the coming months:

1. **NUMA support**:
   Large multi-core VMs are often NUMA (non-uniform memory access), a.k.a. multi-socket. In other words, subsets of the cpus are closer to parts of the memory. Most of our use cases, including Open MPI and Seastar, make use of various NUMA-related features supplied by the kernel. These kernel features allow the application to query the NUMA configuration, to allocate memory for a specific core, and to pin a thread to a specific core. OSv supports only some of these features, and will need to support the rest. Our recent benchmarking efforts of OpenFOAM indicated that not properly supporting NUMA can slow down an Open MPI application by as much as 20%.
2. **Cloud-init**
   We know that OSv's cloud-init is still missing some standard cloud-init features that we need to complete. See more on this below (in the package management section). Specific requirements are also described in corresponding integration sections.
3. **Performance**
   While much of the work on OSv this year has been to get applications to run, our ultimate goal is to get them to run *efficiently,* and we've unfortunately noticed that in some benchmarks, like a HDFS benchmark we tried for the "Big Data" use case, OSv's performance is today significantly lower than Linux's. Preliminary studies have already been done, but further analysis is required. We are going to debug these use cases, find where the performance degradation comes from, and fix it.
4. **Improving file-system performance**
   In our evaluation experiments we have realised that OSv's operations on the underlying file-system (ZFS) [15] do not perform as well as in Linux. Thorough analysis has revealed limiting factors in the implementation of the Virtual File System (VFS) that essentially serialised all disk access. Preliminary patches have been provided, significantly improving the performance, however additional experimentation is

required to improve the performance even more. We are also evaluating ZFS [15] against other file-systems, more typical in Linux environments.

5. **Virtual RDMA**
   We mentioned virtual RDMA in the previous section (on the hypervisor), but in the next phase we will also need to extend it into the OSv guest, to make sure the guest-side driver is working properly on OSv, and working efficiently.

6. **Improving NFS integration**
   For HPC environments access to shared storage is a mandatory requirement. Current NFS integration into OSv already offers quite comprehensive support for NFS. Additional experiments are required to test various potential scenarios of using shared storage through NFS. Furthermore, performance analysis must be conducted to evaluate and compare with existing systems. Recently HPC centers have also been actively pushing the LUSTRE [16] file-system which is particularly suitable for high performance and big data accesses. LUSTRE is a massive project, however its client API should be analysed for potential integration into OSv.

3.3 Seastar

In the previous version of this document, D2.16 [2], we presented a detailed introduction to Seastar, a new C++ API for writing efficient asynchronous server applications. In D4.4 [14], we expanded on that introduction by providing an (incomplete) tutorial to writing applications in Seastar.

At the time of D2.16 [2], the basic concepts and architecture of Seastar had already been in place: share-nothing architecture (each core works on different data), single thread per core, futures and continuations. But the implementation was an incomplete prototype. In the past year, the ScyllaDB team has been working hard at making the Seastar framework more mature, more complete, and more efficient. Seastar is very important to ScyllaDB because the company's main product, the scylladb distributed database, uses Seastar, and Seastar is its "secret sauce" for achieving 10 times the performance of its competitor (Cassandra).

Our goal is to develop Seastar as a general-purpose library which will make it possible to write many different highly-efficient server applications for the MIKELANGELO cloud, and not just the scylladb application. This means improving Seastar's documentation, improving its APIs, addressing the needs of users outside of ScyllaDB, and more. The "Cloud Bursting" use case of MIKELANGELO will demonstrate the benefits of using Seastar to develop server applications of the future.

The following Seastar requirements arose from the MIKELANGELO project this year, and were addressed in WP4:

1. **Seastar documentation:**
   We believe that good documentation is essential for the adoption of Seastar both inside and outside MIKELANGELO. Seastar's APIs are significantly different from the more widely familiar Linux APIs (such as sockets, read(), write(), etc.), and also require significant C++14 familiarity. So we believe that without good documentation, we might scare away potential users outside its current circle of developers. This is why we spent significant effort to document Seastar in various forms: On its website and wiki (to outline its architecture, and so on), by writing a tutorial in book form, and API documentation (in doxygen form). All this documentation is freely available on the web [17], just like Seastar itself. This effort is only partially completed (e.g., the tutorial is currently only 26 pages long), and we believe it is important to continue this effort.
2. **Remote Procedure Call (RPC) in Seastar:**
   Seastar runs inside a single machine, but many Seastar applications, including the scylladb distributed database which we use for the "Cloud Bursting" use case, offer a distributed service and therefore need convenient primitives for communicating between different machines running the same application. We designed and implemented RPC capabilities for Seastar, where the application running on one machine can call a normal-looking function returning a future value, while the Seastar seamlessly communicates with the remote machine, runs the function there, retrieves the result, and resolves the previously-returned future with that result.
3. **Seastar I/O scheduling:**
   One of the key requirements that arose in the "Cloud Bursting" use case was to ensure that performance did not deteriorate significantly during a period of cluster growth. When a Cassandra cluster grows, the new nodes need to copy existing data from the old nodes, so now the old nodes use their disk for both streaming data to new nodes, and for serving ordinary requests; It becomes crucial to control the division of the available disk bandwidth between these two uses. For this, we designed an I/O scheduler for Seastar: The application can tag each disk access with an I/O class, for example a "user request" vs. "streaming to new node", and can control the percentage of disk bandwidth devoted to each class.
4. **IOtune:**
   Seastar's disk API is completely asynchronous and future-based just like everything else in Seastar. This means that an application can start a million requests (read or write) to disk almost concurrently, and then run some continuation when each request concludes. However, real disks as well as layers above them (like RAID controllers and the operating system), cannot actually perform a million requests in parallel; If you send too many, some will be performed immediately and some will be queued in some queue invisible to Seastar. This queuing means that the last queued request will suffer huge latency. But more importantly, it means that we can no longer ensure the

desired I/O scheduling, because when a new high-priority request comes in, we cannot put it in front of all the requests which are already queued in the queue beyond Seastar's control.

So clearly Seastar should not send too many parallel requests to the disk, and it should maintain and control an input queue by itself. But how many parallel requests should it send? If we send too few parallel requests, we might miss out on the disk's inherent parallelism: Modern SSDs, as well as RAID setups, can actually perform many requests in parallel, so that sending them too few parallel requests will reduce the maximum throughput we can get in those setups.

The requirement to tune the I/O parallelism to what the disk can actually handle led to the development of "IOtune", a tool that runs on the intended machine, tries to do disk I/O with various levels of parallelism, and discovers the optimal parallelism. The optimal parallelism is the one where we get the highest possible throughput, without significantly increasing the latency. This is the amount of parallelism which the disk hardware (and RAID controllers, etc.) can really support and really perform in parallel. After discovering the optimal parallelism, IOtune writes this information to a configuration file, and the Seastar application later reads it for optimal performance of Seastar's disk I/O.

5. **Seastar threads and scheduling:**
   Seastar applications do not use standard OS-level threads, instead using much lighter-weight continuations. But there is a lot of convenience in programming with threads, so we added the concept of "Seastar threads". These are not Posix threads but rather continuations which have their own stack, and switch out every time a future is waited-on with the "get()" method (Seastar threads are not preempting). Seastar threads also support rudimentary control over the amount of CPU time which each thread gets: a Seastar thread can yield if it receives over a threshold percentage of the CPU. This feature is useful to guarantee that low-priority background tasks in the server cannot monopolize the CPU.
6. **Miscellaneous performance improvements:**
   We discovered and fixed several places where the Cloud Bursting use case was showing reduced performance because of inefficient code. For example, we improved networking performance by implementing better batching, improved asynchronous disk I/O, and reimplemented output streams (used for both disk and network I/O).
7. **Encrypted sockets:**
   The Cassandra re-implementation for the "Cloud Bursting" use case needed support for encrypted sockets (TLS, what was formerly known as "SSL"), to allow clients to communicate securely with the server. So we added support for it in Seastar.

3.3.1 *New Requirements for Seastar*

For the next milestone, we expect that we will continue to devote additional effort to some of the above-mentioned requirements (e.g., the documentation). But we already encountered several additional requirements which we will need to address:

1. **CPU scheduling:**
   The I/O scheduling feature which we described above (and implemented this year) goes a long way to ensure that when the system needs to run two unrelated tasks (such as the data-streaming and the request-handling mentioned above) one of them does not monopolize all the resources. However, in some cases scylladb has work in which there is very little disk I/O but significant computation - and in such cases the I/O scheduler is not enough and we need an actual CPU scheduler: We need a mechanism of tagging continuations (Seastar's unit of computation) with a class, and determining what percentage of the CPU each class may get, and which of the continuations to run next. The hard part is how to do this efficiently, without hurting the very low overhead of Seastar's continuation and its trivial continuation scheduler (which today just keeps a list of ready continuations, and runs them in order).
2. **Scheduler for blocking system-calls:**
   For efficiency, Seastar runs just a single thread per core. This means that this thread must never call any system call which might block, because a blocking system call would mean the core remains unused for a period of time. For this reason Seastar uses non-blocking network I/O, and asynchronous I/O for disk I/O.
   However, there are unfortunately a few system calls in Linux which have no non-blocking or asynchronous alternative. One of these is unlink(), for deleting a file. Deleting a file can be slow for large files, as the amount of work it has to do is proportional to the length of the file. Unfortunately, there is no system call which only starts a deletion and only later notifies the caller when the deletion completed. Another example of a blocking system call is fsync(), needed when the program needs to know when a certain write actually reached the disk.
   Because of these blocking system calls, Seastar keeps a global pool of additional threads, beyond the main one-thread-per-core, just for running these blocking syscalls. A Seastar application may want to run several of these blocking system calls in parallel, so we need to allow multiple such syscall threads to exist, but for obvious reasons we need a limit on the number of these threads. This limit creates a scheduling problem: If one component of the system starts 100 of these blocking system calls, and then a second component tries to run just one blocking system call, it will see huge delay, as it will need to wait for many of the previously-submitted system calls to complete, So just like we did for disk I/O, we also need a scheduler for

these blocking system calls, to ensure that two components have fair access to the system call threads.

3. **Multi-tenancy:**
   Perhaps the "holy grail" of the Cloud Bursting use case is multi-tenancy: If a very large cluster is shared by several tenants (database owners), there are some unused or underutilized nodes, and one of the tenants suddenly needs more machines to handle its load, their data is already spread out over the very large cluster, so the cluster can almost instantaneously handle a much higher load using the spare capacity. This is why support for multi-tenancy in Seastar is important for MIKELANGELO.
   A multi-tenant application needs better *accounting* of how much work each tenant does: CPU usage, disk usage, network usage, and memory usage. It also needs better and stricter ways of limiting how much of each of those resources is used to service any particular tenant. Beyond the I/O, CPU, and blocking system call scheduling that we already mentioned, we will need to add additional features, such as separate memory pools for each tenant, network I/O scheduling, and more.
4. **Improve power consumption:**
   Seastar started as a polling-only event loop, but today already has a "sleeping" mode where Seastar can go to sleep if there's nothing to do. The sleeping mode needs to be further improved, to further reduce power consumption on relatively idle machines, without hurting latency.
5. **Improve monitoring capabilities:**
   Seastar already supports exporting statistics via collectd, but we need to export many more statistics, and also support dynamic selection of reporting frequency. Finally, we should tie this collectd monitoring with MIKELANGELO's monitoring, developed in WP5.
6. **Seastar on OSv:**
   In principle, Seastar runs on any Linux-like OS, and in the past we ran it on both Linux and OSv. However, as Seastar evolved, it started to use some Linux features which are not available on OSv. To make it again runnable on OSv, we need to add the missing features to OSv, and/or make it optional in Seastar to use those features. The biggest missing feature is asynchronous file I/O: for disk I/O, Seastar relies on Linux's asynchronous I/O ("aio") APIs, and those require support in the file-system implementation. In Linux, only the XFS file-system fully supports asynchronous I/O - but on OSv, our only file-system (ZFS) does not. A better alternative to adding an asynchronous file-system to OSv is adding an asynchronous file-system to Seastar:
7. **Seastar file-system:**
   Seastar makes it easy to write asynchronous code, so it is easier to write an asynchronous file-system in Seastar itself, rather than in the OSv kernel. Moreover, implementing the file-system in Seastar will have additional advantages: We can make

it more efficient because we will implement the semantics we really want, not necessarily POSIX semantics. We will ensure that all operations (including file delete, for example) are asynchronous, and design the file-system to minimize communication between cores (e.g., by assigning different parts of the file-system to different cores). We will also have better control over the parallelism of the requests (as explained above regarding IOtune). Finally, a Seastar file-system will be usable on both OSv and Linux, so it would not require Linux users to switch to the (currently) unpopular XFS file system.

3.4 Application Management Tool

The package and application management tool in MIKELANGELO project is based on Capstan [18]. Following a detailed analysis of existing tools for building and managing OSv-based virtual machine images, we designed a preliminary architecture enabling the required functionalities. This architecture (Figure 5) was initially presented in D2.16 [2].

The MPM (MIKELANGELO Package Management) architecture assumes three layers. The bottom layer represents the package repository storing application packages and their metadata. Because multiple repository providers are envisioned, an abstraction of a package repository is required. The middle layer is the core of the MPM and provides a set of functionalities for building, validating and composing application packages and runnable virtual machine images. The topmost layer (MPM Client) is a user-facing component that allows interaction with the remaining two layers. It is currently implemented as a command line interface (CLI).

Components in Figure 5 marked with green color, namely the Package Builder and the Image Composer, have been the main focus for implementation during the first 18 months of the project. The Package builder component is used to create new packages and store them in a central repository. Each package may be comprised of application binaries, supporting libraries and additional configuration files. On the other hand, the Image Composer component uses application images to build a self-contained runnable application virtual machine image. This image is ready to be executed in the target environment, either locally with the use of the MPM client or on a public/private cloud.

The two yellow components (MPM Client and MIKELANGELO Repository) have received updates necessary to support new core components:

- MPM Client (extension of Capstan) now supports package builder and image composer. Preliminary support for OpenStack provider has also been added to the client;

- MIKELANGELO Repository has dedicated storage for application packages from where they are used by the image composer.

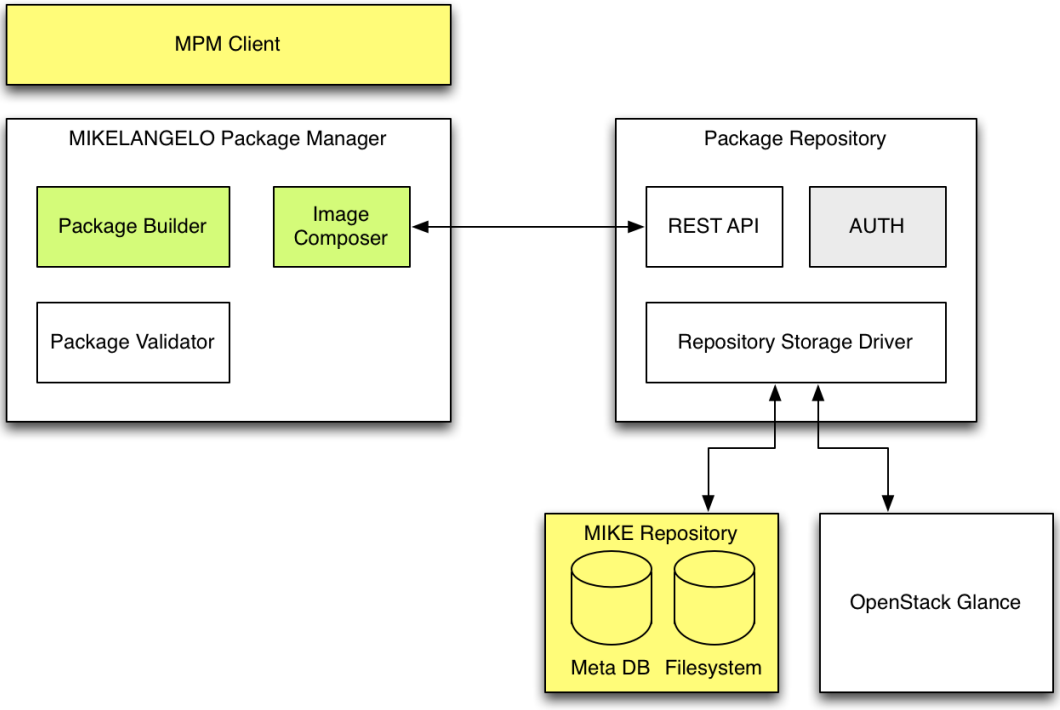

Figure 5. Initial architecture of the MIKELANGELO Package Management. Components marked with green color represent the main development focus.

Besides these changes, we have also started on the integration with OpenStack, focusing first on the ability to push composed images into Glance (image service) and then running them with Nova (compute service). Although this integration is only preliminary, it already demonstrates the benefits of automation of building and running OSv-based applications. Users are no longer required to build (compose) images, upload them into OpenStack Glance manually and configure through graphical user interface that is overly complicated for the task at hand.

These changes are already improving the management of the lifecycle of OSv-based applications. However, the initial evaluation done by some consortium partners in WP6 and reported in deliverable D6.1 [6] revealed several new requirements for a more powerful application management tool. These new requirements are presented next, together with original requirements that have not been resolved yet.

1. **Elimination of external dependencies (initial requirement).**
   Capstan currently still relies on QEMU/KVM to compose and configure target virtual machine images. Currently this is not a mandatory requirement because users are still

using Capstan in controlled environments, but the requirement is very important for external users.

2. **Change of the underlying architecture (initial requirement).**
   While we were implementing initial support for the chosen cloud provider, it became even more evident that the current architecture of the Capstan tool is not suitable for supporting alternative providers nor for providing services to external integrators (for example HPC integration). Because our main target so far was simplification from the end user's perspective, we are planning to address this requirement in the next iteration.
3. **Support for other guest operating systems (initial requirement).**
   We have reevaluated this requirement with other potential systems, in particular unikernel systems. It became evident that there are significant differences in the way images are composed and/or compiled. Consequently, we are postponing this requirement for now. In D6.1 [6] we also mention a new project (Unik [19]) that has been trying to address this. We are following the project closely, and in discussions with the team behind it about potential collaboration.
4. **Run-time options (initial requirement).**
   This requirement is going to be addressed in the next iteration. It is important to allow users to work with OSv-based applications as if they were simple processes. During the comprehensive benchmarking of OSv applications it was observed on several occasions that it was impossible to understand the application without looking at the actual code. To this end we are going to expand the package metadata capabilities allowing package authors to provide reasonable documentation as well as the intended use cases (for example, default commands).
5. **Finalise cloud integration.**
   The initial target is to fully support OpenStack integration with more features available out-of-the-box. When we move from application packages, into runnable instances it is essential that their lifecycle can be managed from a central place. OSv applications are not typical in that the user can seamlessly connect to a remote machine to reconfigure it. Besides improving support for image and compute services, additional services are going to be integrated: networking and storage. This is going to be addressed in the next iteration.
6. **Abstraction of cloud provider.**
   We believe that relying on a single cloud framework will limit the exploitation potential of the application packaging. To this end we are already planning to abstract the concept of a cloud provider. An implementation for Amazon AWS (and potentially OpenNebula) is going to be provided with this improvement.
7. **Package hub.**

Users are currently required to download and install the package repository locally. For the time being the number of packages and consequently the size of the repository is not large, so a central hub of all packages has not been necessary. With an increasing number of applications and packages, such a centralised hub will simplify the workflow acquiring only packages required by the end user.

3.5 Application Packages

Pre-built application packages are another important contribution of WP4 when it comes to an overarching technology stack of the MIKELANGELO project. Up until now, the following packages have been provided by the consortium:

OSv launcher (OSv kernel). A bootable image containing the OSv kernel and the tools required to compose the target application image. This image is inherently included in all application images whenever they are composed. A bootstrap package accompanies the launcher further providing system-wide libraries that are required for proper operation of the OSv-based VM (for example, a library for the ZFS file-system [15]).

HTTP Server. The package contains the REST management API for the OSv operating system. It allows users to query and control all kinds of information about the OS.

Command Line Interface (CLI). Provides a simple interactive shell for OSv virtual machines, mostly useful for debugging purposes. The CLI primarily offers a user-friendly interface to the HTTP server. Therefore, it automatically includes the HTTP Server package.

Open MPI [20]
This package contains libraries and tools required to run MPI-based applications. It is intended as a supporting package for HPC applications that need the MPI infrastructure: either just Open MPI libraries or also the *mpirun* command for launching parallel workloads.

OpenFOAM Core [21]**.** This package includes the base libraries, tools and supporting files that are required by arbitrary applications based on OpenFOAM toolkit.

OpenFOAM simpleFoam. One particular OpenFOAM solver application, used by the Aerodynamics use case. The package only contains a single binary (application) as all the remaining libraries are already available in the core package. Similar application package contains rhoSimpleFoam application solver, however it has not been thoroughly tested using the aforementioned use case.

Java [22]**.** The package contains an entire Java Virtual Machine and supporting tools for running arbitrary applications on top of OSv. It has been successfully tested with several applications such as Cassandra, Hadoop HDFS and Apache Storm.

Cloud-init [23]**.** This package allows contextualisation of running instances, for example setting the hostname of the VM and creating files with specific content. If this package is included in the application image, it will, immediately upon start, look for various data sources used in common cloud environments:

- Amazon EC2 [24]: used by Amazon AWS as well as default OpenStack. It provides a metadata service that cloud-init [25, 26] can access at a specific IP address.
- Google Compute Engine [27]: used in cloud, provided by Google
- No cloud-like data source [28]: in this case user data is attached to the target VM as a dedicated disk image from where the cloud-init will read and load necessary data.

Hadoop HDFS [29]
Hadoop HDFS is one of the core components of the big data application. It is a Java-based application that inherently uses subprocesses for various internal tasks. This package was chosen to demonstrate the required steps for patching such applications and making them compatible with the OSv unikernel.

Along with these application packages we are providing several demo application packages used in the tutorial for the MIKELANGELO Package Management tool. These will help newcomers to quickly grab a working demo application and start making their own apps run on top of the MIKELANGELO stack.

Application packages described above already allowed various experiments with the Aerodynamics use case, including running them in the OpenStack-powered cloud as well as HPC environment, driven by Torque PBS [30]. Consequently, from the perspective of this use case, we can attest that the requirements related to OpenFOAM have been fulfilled. The following is a list of most important pending or newly discovered requirements related to the use cases and application packages.

1. **Cloud-init module must support attaching of network shares.**
   Standard cloud-init supports the *mounts* option allowing users to provide specific volumes and network shares to be mounted into the target VM automatically upon start. For HPC applications it is mandatory that such support is integrated into OSv's cloud-init module allowing execution of experiments based on external data.
2. **Customisable Open MPI application package.**
   The existing Open MPI application package must be extended to support additional configuration options suitable for wider ranges of MPI applications.
3. **Support for *mpirun* command.**
   HPC use cases in MIKELANGELO project (Aerodynamics and Cancellous Bones) rely heavily on the MPI for distribution of workload and synchronisation between individual processes. *mpirun* is the main entry point for these two applications. Because the OSv-based infrastructure is fundamentally different from the one in a

typical HPC setting, the MPI application package must ensure transparent use of virtualised workloads running in OSv. This requirement is also fundamental for external exploitation in HPC applications. The current implementation of the mpirun command is a link to the orted command present in the Open MPI library and it is used to create the MPI environment and execute the MPI application.

4. **Additional use-case specific pre-built application packages.**
   This is a placeholder requirement for all the remaining applications that will be running OSv-enabled application images, in particular for the Big-data and Bones use cases. All of these packages need to be prepared in a way suitable for a broader audience. This will allow reuse of composable, pre-packaged applications, and seamless deployment in target environments.
5. **Multiple runtime environments.**
   Besides Java we are also interested in other runtime environments, such as Node.js, Go and Python. Node.js has already been tested, but not provided as an application package. Other environments are going to be evaluated and integrated accordingly. This requirement has not been expressed by any of the use cases. However, following discussions with potential external stakeholders it is vital that they are supported with little or no modifications required to user's applications. Very minor and mostly boilerplate changes should be required at most.

4 The Intermediate MIKELANGELO Architecture

In this section we discuss the progress regarding the overall MIKELANGELO architecture. In contrast to the previous sections, here we take a holistic view describing how all separate components are integrated into one system. Although many components integrate in the same way in any field, there are some peculiarities for our two main fields of application: cloud computing and high performance computing. Thus, this section first provides a generic overview on the integration of individual MIKELANGELO components, before it splits the discussion into cloud- and HPC-related subsections.

The goals of providing an integrated architecture for MIKELANGELO are ease-of-use, optimization for added value, and focused research. Ease-of-use aims for an integrated platform that fits nicely into common cloud and HPC infrastructures. Thus, it should be simple for admins to use all or just some parts of the MIKELANGELO stack. Optimization aims to leverage individual optimisations as well as cross-layer optimizations to maximize the added value of MIKELANGELO's advancements. Focused research refers to leveraging a tight integration of components to identify new research opportunities and to discover and resolve inefficiencies.

The overall progress of the intermediate architecture over the initial architecture entails more detailed design as well as progress on the technical integration. In this iteration the original architecture has been refined based on new findings, a better understanding of all components, and progress of individual components. The actual integration of components is even more important as it shows that the conceptual integration is viable. Both for the cloud and the HPC stack significant progress has been made, such that related stacks are running in the GWDG cloud and the USTUTT HPC testbed. As of writing this report application tests are being run regularly on those infrastructures to verify the functionality of components and to assess their performance.

In the following subsections we provide details on the goals of the intermediate architecture, considerations for cross-layer optimization, and the progress on the cloud and HPC stacks.

4.1 Goals

The main goal of the MIKELANGELO overall architecture is, as already pointed out, a common software-stack for cloud and HPC environments. The two architectures are used for the validation of principal concepts addressed in the MIKELANGELO project. They furthermore serve as examples for additional integrations done by external integrators.

Besides improving the overall performance of the virtual environments, MIKELANGELO seeks to enable developers to package their applications ready for execution, regardless of the

underlying infrastructure (Cloud/HPC/Desktop PC) used for execution and its actual hardware. The transparency provided by this abstraction opens completely new horizons to users who are no longer required to be concerned with the compilation and execution of their applications. This removes the traditional limitation in the HPC world that users have to use the HPC clusters exactly as provided, meaning they usually cannot request a certain operating system nor specific kernel nor the libraries, giving them the most accurate result. The environment itself is abstracted, too. For example, fast storage for intermediate application data is mounted to a certain directory inside the virtual guests by the infrastructure middleware. Users and developers do not need to consider the actual path/file-system of the physical environment, they just can read and write data to a directory inside the guest - the actual mount paths are defined by the metadata that cluster/cloud admins need to configure for their infrastructure. Applications do not need to be recompiled (with many different compiler options) for each new hardware setup available.

System administrators and devops benefit from a very lightweight configuration of physical clusters and simple deployment of new applications. These are simply packaged as runnable virtual machine images (= a single file) and accessible to their users.

Further, the goal is to benefit in the static, inflexible HPC world from all the advantages that Clouds offers, like live-migration, suspend and resume (= checkpoint and restart), build once and run everywhere.

4.2 Cross-layer Optimization

MIKELANGELO's holistic approach is in a powerful position to provide cross-layer optimization for virtual infrastructures. MIKELANGELO spans a vast amount of the computing stack. Starting in the OS kernel with our hypervisor, sKVM. At the other end of the spectrum we arrive at the application level with a set of use cases. Spanning this broad field, we are able to use high-level metrics, such as QoS indicators to steer low-level mechanisms.

At the heart of MIKELANGELO's cross-layer optimization lies the holistic monitoring framework snap [31]. First, snap offers a plethora of monitoring probes, many of which have been developed specifically in the context of MIKELANGELO. In addition, snap allows dynamic filtering and pre-processing to take place in its data stream. The two major benefits for cross-layer optimization in large virtual infrastructures are a unique, flexible interface, and snap's inherent scalability. In the presentation of the following snap can be seen as permeating the whole stack, due to its cross-layer nature, delivering vital information to controllers at various layers.

In the lowest layers of abstraction, MIKELANGELO tackles components in the host's kernel. Specifically, the hypervisor KVM is extended by three components IOcm, Virtual-RDMA, and

SCAM. Each of those components allows for optimization within a host directly. IOcm will contain a control mechanism to allocate the right amount of IO cores depending on the workload of running VMs. Virtual-RDMA allows to set up efficient shared memory communication between VMs using IVSHMEM as well as utilise fast and low latency network interconnects for efficient communication between different hosts. SCAM will allow the detection of malicious VMs and isolate them or mitigate against their attacks. However, the real benefits of cross-layer optimization will be reaped in the higher layers.

Considering the cloud scenario a controller in OpenStack will be implemented to use the OS-level features mentioned previously. From the perspective of a cluster manager it is possible to change VM allocation to optimize for IO efficiency and security, beyond the capabilities of host-only mechanisms. Thus, in the cloud layer we are going to target optimized IO efficiency across the cloud based on VM placement and configuration. Examples for potential inefficiencies to be remedied are the co-location of heavily communicating VMs to use shared memory communication, the distribution of VMs to avoid hot spots of IO contention, and the isolation and analysis of VMs flagged as malicious.

At cluster-level, VM live-migration will be implemented. Live-migration is the movement of a VM from one physical host to another while the VM is running. The end-user should be unaware of this process when done properly. One of the most significant advantages of live migration is the fact that it facilitates proactive maintenance. If a failure is suspected to be imminent, the potential problem can be resolved before a disruption of the service occurs. Live-migration can be applied also for load balancing, the VMs can be dynamically balanced among new resources granted on demand resulting in less contention between VMs sharing a physical host.

From the perspective of applications further cross-layer optimization is possible. Whereas on the level of the cluster manager there can only be steps taken to make sure that resources do not contend and that IO is handled efficiently, in the application layer we can measure the impact on application performance. As MIKELANGELO's use cases revolve mostly around batch-like computation, the execution time for jobs is an important metric. Considering potential real-time applications in the context of the big data and cloud bursting use cases, latencies may also become important. Intermediate metrics that affect these, such as intermediate component latencies, IO throughput and IPC, can all be measured by Snap as well, and used to help steer controllers.

The development of cross-layer optimization is one of the core outcomes of MIKELANGELO, because it only becomes possible in a completed and integrated platform. Thus, the concrete work on cross-layer optimization will accelerate in the following months, as the base platform integration reaches a level of maturity.

4.3 Integrated Cloud Infrastructure

The progress on the integrated cloud architecture is reflected by a more detailed concept of the integration, especially regarding potential cross-layer optimizations. Furthermore, the actual technical integration in the GWDG cloud testbed has progressed well with several components from the MIKELANGELO stack already running and being used for use case tests. Components already integrated in one form or another include IOcm, OSv, Snap, and the MIKELANGELO Package Manager. Currently, the integration with virtual RDMA and SCAM are pending. During the next phase the architecture will be extended by developing an online scheduler for OpenStack that will leverage MIKELANGELO capabilities. Furthermore, work on optimizing performance is expected to increase further due to the continuous integration efforts in WP6.

The main goal of the online scheduler is to allow adaptive control of the cloud infrastructure. Considering that a large portion of VMs in the cloud is being used for long-running services, online scheduling is of special interest. Currently, OpenStack does not provide an online scheduler, but it relies on initial placement of VMs upon creation. Recent releases of OpenStack provided a tighter integration with libvirt focusing on live-migration. This integration is a key enabler for an online scheduler. The scheduler in turn is required to implement and evaluate the ideas described in the previous section. As of writing this deliverable the scheduler exists as an early prototype. The prototype showcases how monitoring can be tied together with the control features of OpenStack for VM placement. An internal release and evaluation of the scheduler is planned for M22.

In the following subsections, we consider the benefits and hurdles of the MIKELANGELO components in a bottom-up fashion. We start with sKVM's components, IOcm, virtual RDMA, and SCAM. Then we discuss OSv as a guest operating system, Snap for holistic monitoring, and finally MPM for application package management.

4.3.1 IOcm

IOcm can be deployed on any server using KVM to host virtual machines. However, to get the maximum benefit IOcm should be deployed in an environment in which the virtual machines have a heavy I/O load, such as is found in many public and private cloud environments. IOcm is implemented as a set of patches on top of KVM. It does not modify the interface to the hypervisor, and can therefore be deployed on existing KVM installations without any modifications to the guest images. IOcm includes a component that monitors the I/O and CPU loads at the hypervisor level (transparent to all VMs). It can then manage local resources by allocating CPU cores as required according to the measured server load. IOcm works in conjunction with any existing cloud scheduler (such as OpenStack) that knows how to work

with KVM. The cloud scheduler is responsible for the cloud-wide management of resources, and placement of VMs.

4.3.2 *Virtual RDMA*

The implementation of virtual RDMA prototype I is targeted at high speed inter-host network communication based on InfiniBand ConnectX-3 NICs. Meanwhile, this implementation also includes a shared memory component based on vhost-user that can be directly used for intra-host communications. The current GWDG infrastructure doesn't meet the requirement of the InfiniBand NICs, but the shared memory module can be utilized and deployed easily across the nodes. So the Big Data applications can benefit from the improvements of using the vhost-user implementation.

The work to integrate the shared memory component into GWDG infrastructure has been planned for the remainder of project year two. A similar integration design to the one used in the HPC deployment at USTUTT will be used, i.e. the same software packages and same design architecture. As shown inFigure 3, the only implementation difference for GWDG is that vhost-user is combined with normal Ethernet NICs through DPDK and Open vSwitch, and the DPDK Ethernet PMD will be used in this case. The rest of the design remains the same.

However, the integration into GWDG infrastructure has to be realized in a different way, with the help of the OpenStack. The correct versions of Open vSwitch, DPDK and QEMU have to be configured and installed with OpenStack, and scripts for starting and stopping the environment have to be rewritten in Python and managed by OpenStack. More integration details for the GWDG infrastructure will be provided in the later report by the end of the second project year.

The design prototype II is also planned to be implemented in the second project year. In principle the implementation doesn't rely on specific hardware and should support common InfiniBand NICs, which makes it possible to be integrated and utilized by GWDG infrastructure directly. Details about the possibility of integrating design prototype II into GWDG infrastructure and how it could be accomplished will also be described in the later report by the end of the second project year.

4.3.3 *SCAM*

Network security and endpoint security are essential for customers of cloud services due to the shared nature of the cloud. Therefore, cloud providers use a variety of tools and techniques to identify attacks and either prevent them or mitigate their consequences. Most attacks in a cloud environment are similar to attacks in other networking situations. This state of affairs leads cloud providers to rely on well-tested defense technologies such as firewalls,

Intrusion Detection Systems (IDS) and antivirus devices. Unlike these more common attacks, the class of cache-based side-channel attacks is specific to situations in which a legitimate user shares a computing platform with an attacker, a scenario which happens most often in public clouds. These attacks are additionally notable in that they are exceptionally stealthy. There is no explicit communication between the attacker and the target and they do not use any software vulnerability which can either be patched or used to generate a signature for an IDS system. SCAM is specifically tailored to identify these attacks and perform a number of mitigation strategies. As such it is especially suitable for inclusion in cloud environments.

SCAM requires two levels of integration into a cloud environment. At the lower level, SCAM must be integrated with the hypervisor so that it can carry out its functions of monitoring and mitigation. SCAM needs access to such low level resources as hardware counters of cache activity, assignment to specific processors and manipulation of the host page tables. The expected integration of SCAM within sKVM is scheduled for the near future. The higher level of integration is with a cloud infrastructure that is aware of SCAM and can take preventative action across the entire cloud. Commercial public clouds such as Microsoft Azure typically have such cloud-level security response solutions which correlate data on attacks from multiple nodes and respond accordingly, e.g. by isolating suspicious VMs or even stopping their execution. SCAM would fit naturally as one more sensor feeding a security-response center. Integrating SCAM into OpenStack, allowing a security center to receive the readings of SCAM modules, as a future task.

4.3.4 OSv

Typical OpenStack deployment uses the KVM hypervisor through libvirt. Because OSv has been designed specifically for virtualized environments, standalone instances running the OSv unikernel can be executed directly on the OpenStack cloud without any specific configuration. This means that besides being POSIX compliant, OSv is also natively supported in various cloud environments and behaves very much the same as any Linux-based VM. There is, however, some important functionality that OSv does not fully support yet.

First, the current cloud-init [23] support in OSv is incomplete. OpenStack provides cloud-init information to instances using the approach of the Amazon Web Services, namely by providing a virtual server to an instance to query and retrieve metadata and user data. Metadata includes information such as hostname or internal and external IP addresses. On the other hand, user data can be used to configure the instance itself, for example to list the packages to install the first time the machine is booted, create users and their roles, attach secondary disk volumes and run arbitrary user scripts. Most of these features are irrelevant to OSv (or any other unikernel) because the sole purpose of such VMs is significantly different

compared to Linux-based VMs. Nevertheless, the cloud-init module in OSv currently lacks at least the following functionalities:

- Specification of a command to be executed when the instance has finished booting. Command are typically built into the VM image itself which is fine in some case. Other times users should be able to reuse these images using different parameters or even different applications.
- Attaching external volumes to a running instance. OSv should be able to cope with dynamic allocation of new virtual storage devices attached to running instances. Because no interactive shell is available to the user, it must be possible to initialise these volumes (for example to format or resize them).
- Ability to invoke a callback when a VM is finished loading (so called *phone-home* functionality). Sometimes the integrated application must be aware of the fact that underlying machines are fully initialised. Although this can be achieved actively using polling, the proposed way would be to allow for specifying the callback URI.

Additional requirements may be identified for OSv during the implementation of these features and, more importantly, actual use cases (Big Data for example).

4.3.5 *Snap*

The open-source snap telemetry framework [31] is specifically designed to allow data center owners dynamically instrument cloud-scale data-centers. Precise, custom, complex flows of telemetry can be easily constructed and managed at scale. The system is inherently extensible, with a powerful plugin architecture supporting the flexible collection, processing and publishing of metrics. At the time of writing 49 plugins have been open-sourced to address the diverse needs of the data center owner.

Of specific interest in a cloud-scale scenario, the telemetry framework can be managed with one-line commands addressing arbitrarily-sized clusters of hardware - tribes of nodes. Plugins can be dynamically installed, upgraded and loaded at runtime, the catalogue of metrics is thus dynamic, and telemetry tasks (which specify what metrics should be collected, processed and published) can be configured, started and stopped as required. By default, running tasks will automatically invoke newer versions of their plugins if the plugins are upgraded. The execution of a task can even be distributed across multiple nodes if required.

Zero-overhead hardware metric collection is supported: out-of-band metrics can be collected from systems that expose it through special management hardware and interfaces (e.g. through Intel Node Manager using IPMI). Software metric collectors have already been open-sourced that can capture data from local or remote host operating systems (e.g. Linux), hypervisors (e.g. via libvirt), guest operating systems (such as OSv), popular VM and container

management stacks (including OpenStack and Docker), and popular cloud network, messaging, storage, database and analytics middleware.

Once data is gathered, metrics can be filtered or transformed before distribution to minimise the transmission of unnecessary data. Moving averages can be calculated, metadata can be added, and a plugin to automatically increase data resolution around statistically significant anomalies using the Tukey method is nearing publication. In initial tests this has reduced the data transferred by an order of magnitude, without affecting the statistical value of the information transferred.

Metrics gathered by snap can be published to many tools cloud data-centre owners already use extensively. These include InfluxDB, PostgreSQL, MySQL, HEKA, HANA, Kafka, KairosDB, OpenTSDB, RabbitMQ, Riemann and CloudWatch. Powerful analytics tools such as the Trusted Analytics Platform, and dashboards such as Grafana, can thus be easily leveraged.

Security is a fundamental concern in the cloud. Snap has digitally-signed plugin verification switched on by default, API invocation is through SSL, and all payloads transferred between components can be encrypted.

On a more practical note, the scale of modern cloud deployments demands management tools that can be easily integrated into DevOps configuration tools and scripts. The CLI of snap is complemented by a RESTful API to support flexible remote management.

Looking forward, INTEL will imminently release functionality that automatically reduces the resolution of published data for metrics while they are stable, significantly reducing overall telemetry system overhead without reducing statistical usefulness of the data. An additional plugin will be released to capture useful high-level metrics: Utilisation and Saturation data for system components including CPU, network and storage. These metrics are designed to allow very fast identification of resource bottlenecks across a cloud-scale data centre deployment.

The use cases are also generating valuable requirements which are being prioritised for resolution. These include the ability to dynamically adjust the list of metrics being monitored by snap, and the collection of additional metrics from middleware (e.g. sKVM IOcm, HDFS), network infrastructure (e.g. Open vSwitch) and hosted applications (e.g. ScyllaDB).

To facilitate performance analysis, techniques to automatically discover the impact of changes will also be investigated. The possibility of leveraging tools like Trusted Analytics Platform [32] will be explored.

4.3.6 *Application Package Management*

It is seldom possible to execute unmodified Linux binaries directly in OSv. Having ready to run application packages is therefore one of the essential requirements driving the potential uptake of this novel system. Existing approaches to building application images on top of OSv unikernel have required either a complete development environment properly set up or using fixed prebuilt images that can be augmented with additional data.

Package management of the MIKELANGELO project builds upon an existing tool (Capstan). It introduces a concept of an OSv application package that can contain arbitrary binary, source and configuration files. It further enables a workflow to compose these packages into runnable OSv-based virtual machine instances. While the initial development has focused on a standalone tool for managing local applications, we have also started working towards integration with the cloud middleware. The tool released as part of the first official MIKELANGELO release offers preliminary support for deploying composed images into the OpenStack image service (Glance) and creating OSv-based instances using OpenStack compute service (Nova). These integrations already enable users to interact with their public/private cloud provider transparently. Users no longer need to be aware of the underlying middleware and can run their applications locally (for example using the QEMU/KVM hypervisor) or on dedicated infrastructure.

One of the biggest limitations of the current approach is that applications are still composed into virtual machine images locally. This not only requires users to have virtualisation software (QEMU/KVM) installed but also reduces the flexibility of application management. These images, once built, are rigid and cannot be resized due to the underlying structure of the file system. This means that if a user needs another instance with a slightly different configuration (for example because the cloud provider uses different flavors), it has to be recomposed from scratch. We are thus investigating an approach to compose VM images only upon request from the cloud or HPC middleware. OpenStack is enabling this through a project called Murano [33]. Murano treats applications as packages that are dynamically built, deployed and configured (contextualised) for the user based on their requirements. In doing this Murano deploys a lightweight agent onto target virtual machines that deploys required packages and configures them. In the context of a unikernel the approach will need to run the agent as a service as part of the cloud middleware. Using Capstan the agent would configure required virtual machine images and ensure they are deployed on required infrastructure. Despite the fact that a significant amount of Murano capabilities are not applicable to the unikernel ecosystem, users are going to benefit from the fact that their applications are deployed dynamically using only the infrastructure provided by their cloud.

The implementation of the aforementioned idea is going to be as transparent as possible and, preferably, separated from Murano itself, mostly due to the inherent differences between unikernel and Linux instances. We are also looking for a more general purpose solution that can be integrated both into other clouds and with other unikernels.

4.4 Integrated HPC Infrastructure

The integrated HPC infrastructure is evolving towards a common software stack for both, Cloud and HPC as much as is possible. Within the last reporting period the integration of the first version of components developed in the MIKELANGELO project took place. Further, the overall HPC architecture was revised and many new functionalities have been implemented. Some of the new features are based on the requirements that have been identified during the integration work, while most are addressing requirements from the design of the first HPC architecture.

The MIKELANGELO HPC integration is based on the Portable Batch System (PBS) open-source fork called Torque [30]. Torque is a resource manager and simple scheduler for HPC environments, developed by Adaptive Computing Ltd. Torque manages compute nodes and other IT resources, like GPUs or software licenses. Users submit their batch job scripts to Torque's server, the job is then put into a queue and scheduled for execution. Once it is determined that a particular job is ready for execution, it is deployed and executed on a set of exclusively allocated resources. The Torque batch system software is extended to allow users to run their HPC workloads in predefined customized virtual environments - independent of the actual mount-points, software, operating system and hardware in place.

In the following subsections, the evolved HPC architecture is described in detail as well as the current state of the component integrations. Common parts for standard Linux and OSv guests are highlighted. Furthermore, the underlying operating system specific parts, independent of the actual guest OS, are described.

4.4.1 Torque Extensions for Virtual Machines in General

The general extension that allows the running of batch job scripts in virtual, customized environments requires extension of Torque's workflow for the management of virtual environments. This is achieved by several wrapper scripts that take care of the creation and destruction of the requested virtual environments.

Torque is extended in such a way to allow the submission of standard batch jobs, executed on bare metal HPC hardware, as well as in virtual, customized and/or predefined execution environments. These extensions are fully transparent to the end user and offer convenient access to typical features such as:

- in-line resource requests in a job script's header section,
- interactive jobs,
- automatic data staging of job script,
- setting the working directory (usually a path in user's `$HOME`),
- the `STDOUT` and `STDERR` outputs stored in files named like the job and placed in the submission directory.

These are all tools and solutions that are commonly used in HPC environments.

In the following subsections all parts of the MIKELANGELO extensions for Torque are described and explained in detail, as well as its enhanced workflow and the developed MIKELANGELO components integrated. Further details about individual files or parameters and environment variables can be found in Appendix A.4-A.8.

4.4.1.1 Scripts for VM-based Job Execution

There are several files required for enabling Torque to execute batch jobs in virtual HPC environments. In the figure below there is an overview of the files, their purpose and relations.

The green and orange colors in the figure below are referring to the execution level of the scripts. Green refers to scripts that are executed with standard user rights, while the orange scripts are run as root user. Grey doesn't have a meaning besides grouping.

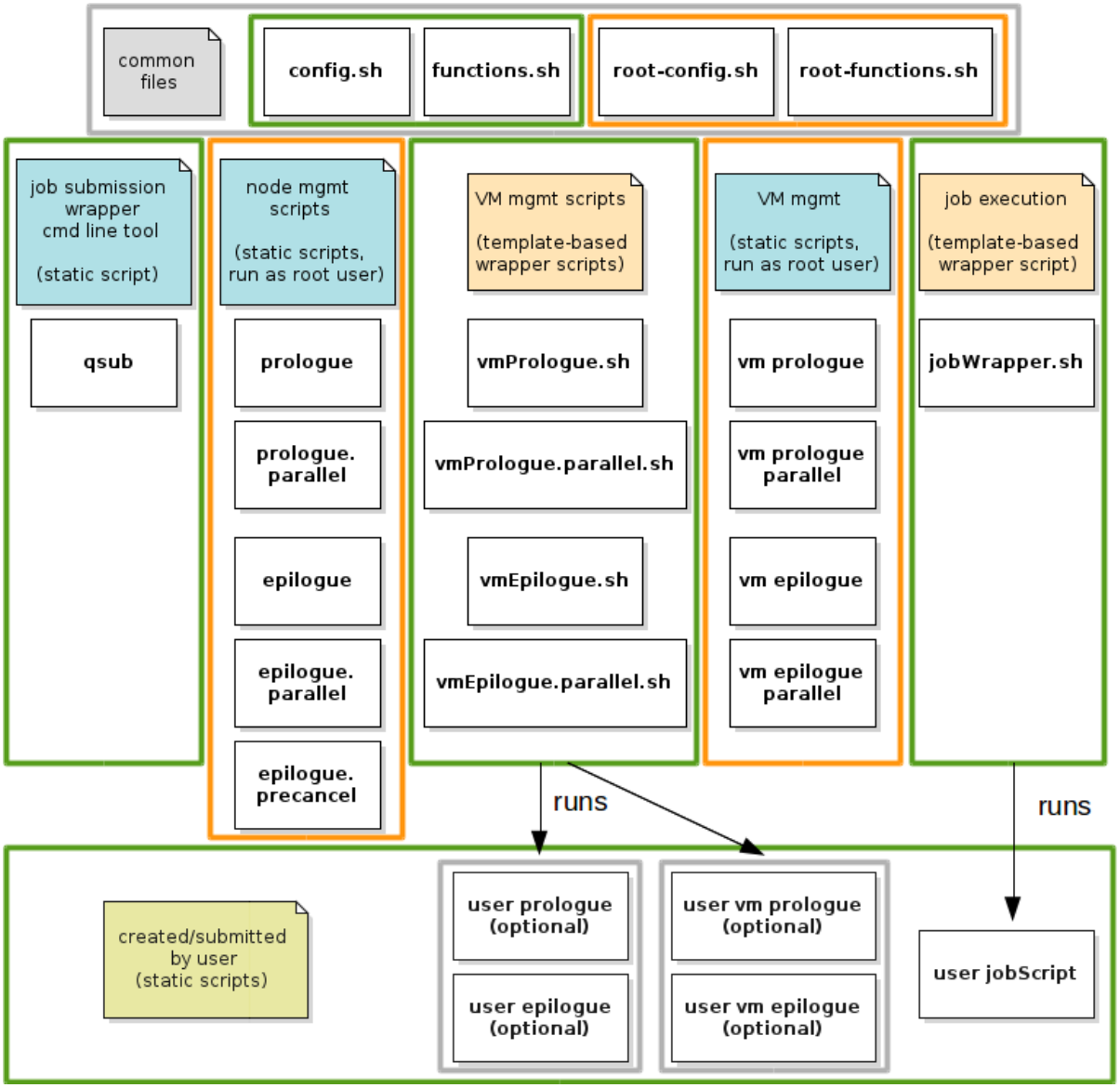

Figure 6. scripts, templates and configuration files.

The boxes in the figure above refer to a disjunct set of files that serve a certain purpose:

- The "**common files**" are used by all other scripts.
- **Job submission wrapper** cmd line tool, wrapps Torque's `qsub`.
- The **node scripts** prepare and clean up the physical nodes
- The user level **vm mgmt scripts** generate files on the nodes that are required for the virtual guest instantiation and cannot be created earlier. Furthermore, they boot the virtual guests in the prologue sequence.
- The root level **vm scripts** are the counterpart to the root level **node scripts**. They are intended for cluster administrators, only. These scripts are run after all other boot processes within the virtual guest have finished, but still before the SSH server becomes available.

- The **job execution wrapper script** sets up the virtual job environment inside the guest and starts the actual user job script within the guest.
- The **user scripts** are generated and submitted by the user, prologue and epilogue scripts are optional

4.4.1.2 Misc Files

In addition to static script files and template based ones, there are some more files that are part of the MIKELANGELO HPC software stack. Some of them are related to the virtual guest's behaviour and their actual environment (kernel, libraries installed, etc), while others are dedicated to the physical machines that host the virtual ones.

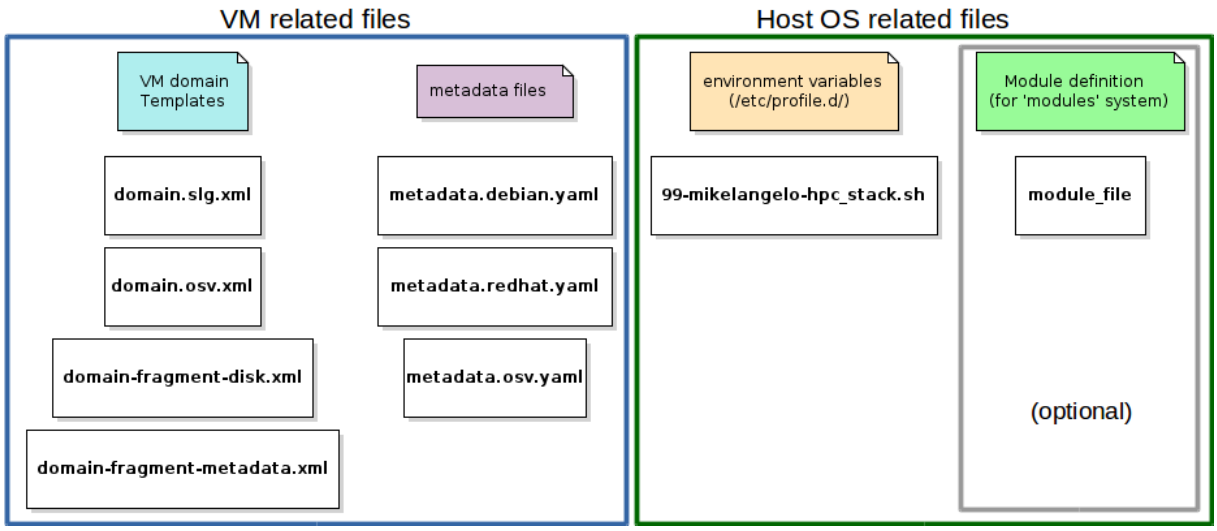

Figure 7. Misc files for VM based job execution.

VM related files

For the virtual guests there are template files that are common for all guest operating systems, and some that are distribution specific.

Host OS related files

For the physical HPC environment, the compute nodes and front-ends, there are also specific files.

4.4.1.3 Components Integrated

Since the prototype version from year one - already able to execute jobs in virtual machines - many extensions of the functionality as well as many new features have made it into the code. One of the major achievements since the first reporting period is that now most of the components that form the MIKELANGELO software stack are integrated in and available to both HPC and Cloud environments.

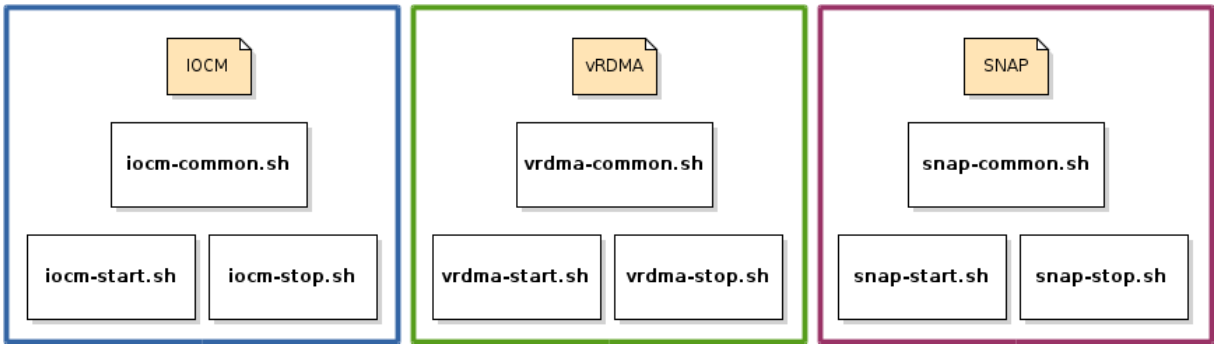

Figure 8. Components integrated into virtualised HPC.

Each of the components is integrated with the help of three script files. The first one, the file '`*-common.sh`', contains functions and all configurations for the component. It is referred to by both the '`*-start.sh`' and '`*-stop.sh`' scripts. These start and stop scripts are executed with root rights in the prologue and epilogue scripts run on the bare metal nodes, before and after a job runs.

Further details regarding the technical integration are highlighted in the workflow section.

sKVM / IOcm

The first version with static core management is integrated, tested and working.

sKVM / Virtual RDMA

Prototype 1 is integrated, tested and currently working with a reduced domain XML definition, only.

Monitoring / Snap

Monitoring is in place, a selection of metric plugins has been successfully tested, but the current setup is not very stable. Reasons seems to be to a major issue with too many open files on the client side of the InfluxDB database. If the issue (which appears to be limited to the InfluxDB client) cannot be fixed, alternatives to InfluxDB will be considered.

Security Module / SCAM

The module is currently not integrated since standard batch jobs have exclusive allocations on the physical nodes. It is not relevant even when using Torque's NUMA domain feature where users have their dedicated NUMA domain, but are still sharing a physical node. In the latter case there is no shared L1/L2/L3 cache and no shared RAM that can be used to attack another user's VM, [40]

4.4.1.4 VM Parameter Extensions for Job Submission

For the virtual job execution users have the opportunity to define many resources, but are free to skip most of these. Parameters needed, but not provided, will be set to the default values that are defined in a global configuration file and are chosen by the HPC system administrators.

Job execution in virtual guests is provided on top of Torque's standard functionality. That starts with the resource request itself. The resources/parameters that are dedicated to the virtual guest(s) are appended to the physical resource request in a similar way. The standard `qsub` resource requests are issued by prefixing a '`-l`', i.e.

```
qsub -l nodes=2,walltime=00:15:00 jobScript.sh
```

The vm resources and parameters are requested by issuing them with a '`-vm`' prefix, i.e.

```
qsub -l nodes=2,walltime=00:15:00 \
     -vm img=image.img,distro=debian jobScript.sh
```

As an example, suppose the user is requesting 16 cores per node using the virtual environment. Because at least one core (the number is configurable) is reserved for the physical host system and 1-2 additional cores may be dedicated to the I/O core manager (also configurable), the number of available cores for job execution is lower:

```
16 (total cores) - 1 (host) - 2 (max cores for IOcm) = 13 cores
```

The user is informed about this reduction at submission time and is allowed to cancel their job and modify the specification of cores per node and IOcm core count request to match their requirements.

For further details and a full list of supported parameters can be found in Appendix A.3

4.4.1.5 VM Parameter parsing

The parsing of VM related parameters is done is the same order as in Torque. Resource requests defined on the command line override the ones set in a job script's header section with '`#PBS -l <some resource request(s)>`'.

All inline '`#PBS -l <some resource request(s)>`' found are merged into the `jobWrapper.sh` script that is at the end submitted to Torque's qsub command line tool.

In case there are inline '`#PBS -vm <some resource request(s)>`', they will be cut off, since it would confuse Torque and the outcome is not clearly defined.

4.4.1.6 Global and Per User Configuration

The global adminstration config file `config.sh` is accessed by all scripts and allows the control of several aspects of the overall behavior and functionality available to users, as well as path and file name definitions and default values for vm based jobs.

Several values are dynamically resolved and some values defined are constants that shouldn't be touched at all. The relevant ones for cluster administrators are listed and explained below.

Configurable parameters

The parameters in the table below are defined in the file `config.sh` and are intended to be set by the administrators globally.

Default values for VM based jobs

The default values for all mandatory parameters, listed below, are applied if the user does not specify them explicitly. They are also defined in the global configuration file `config.sh`.

4.4.1.7 Environment Variables

There are several environment variables, some are required to be set while others are optional. And some other that control the behavior of the MIKELANGELO HPC software-stack. Few of these have a corresponding global variable, if there is one, it is overridden by the user's environment. For details, please refer to Appendix A.8.

4.4.1.8 Workflow / Sequence Diagrams

The whole workflow for a complete job run contains too many single steps to illustrate it in one sequence diagram. Since there are several sequences in a job's lifecycle, the diagrams are broken down accordingly. Each of the sequences in a job's lifecycle is separately executed by Torque.

Job Submission: Job-Wrapper and Pro/Epilogue script wrapper scripts generation

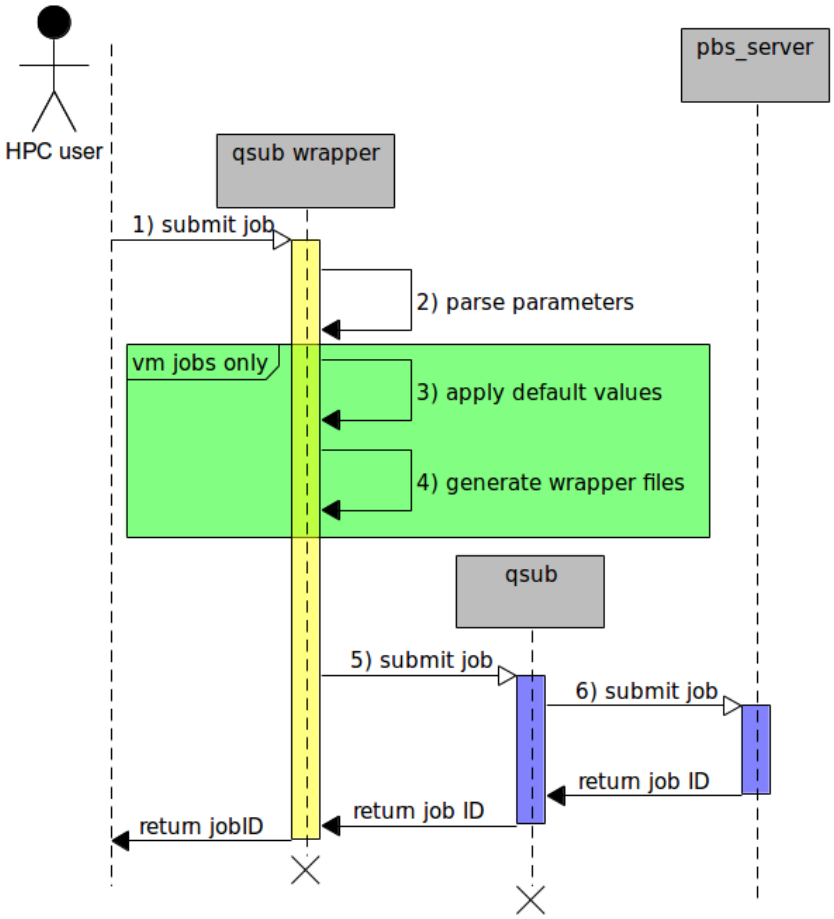

Figure 9. Extended Job Submission Sequence.

1) `qsub wrapper` receives job submission request
2) `qsub wrapper` parses vm parameters and in-line resource requests in the job script
   a) If no vm params found the job submission is passed on the qsub directly, continues with 5).
   b) If vm params are found, continues with 3)
3) `qsub wrapper` applies default values for missing vm params
4) `qsub wrapper` generates `vmPrologue[.parallel].sh` and `vmEpilogue[.parallel].sh` scripts based on templates files
5) `qsub wrapper` submits the wrapper scripts to Torque's `qsub` with same resource allocation as received (i.e. `-l nodes=3:ppn=8,walltime=00:10:00`)
6) Torque's `pbs_server` receives job submission and returns a job id for it

Prologue sequence: Node preparations, VM files generation and Boot process

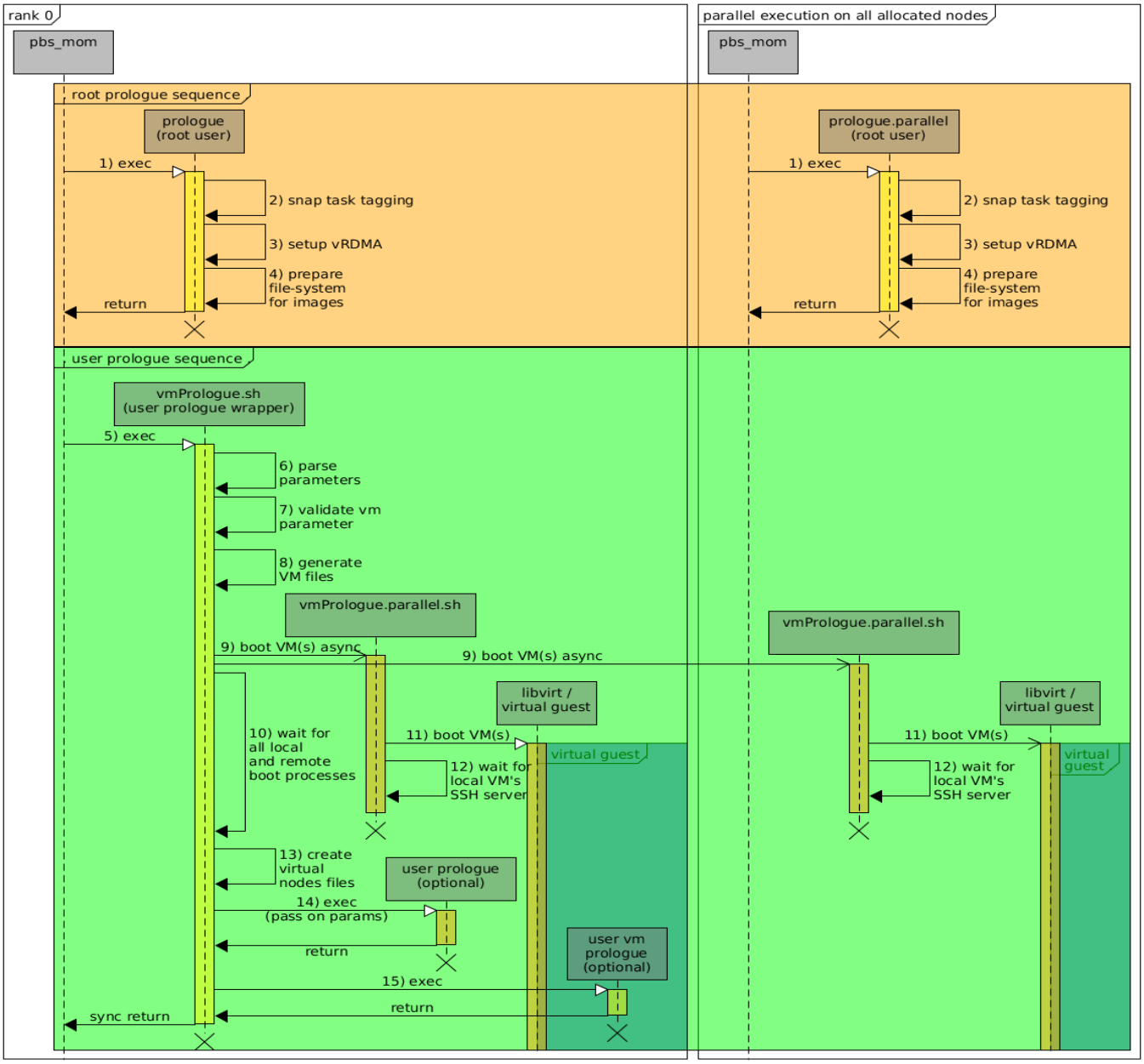

Figure 10. Extended Prologue Sequence.

When the job is deployed on nodes, the next sequence runs.

1) The script `vmPrologue.sh` parse parameters provided by Torque (job id, username, etc)
2) Root `prologue` and `prologue.parallel` scripts sets up vRDMA if enabled
3) Root `prologue` and `prologue.parallel` taggs monitoring task

4) Root `prologue` and `prologue.parallel` create/mounts RAMdisk or checks the shared file-system dir.
5) Script `vmPrologue.sh` validates parameters
6) Script `vmPrologue.sh` prepares nodes
    a) generates `metadata` for each guest
    b) Generates `domain.xml` file for each guest
    c) Copies the images for each guest
        i) If RAMDISK, this happens remotely via ssh
        ii) If shared file-system is used, copying takes place on the first node
7) Boot the VMs on all hosts by calling vmPrologue.parallel.sh for each node allocated
8) Script `vmPrologue.sh` waits until all remote processes (`vmPrologue.parallel.sh`) are finished / flag files are gone
9) On each node the vmPrologue.parallel.sh waits for the VM to fetch an IP
10) Each VM pings its physical host, so the tool `arp` can see the guest's IP
11) Then the `vmPrologue.parallel.sh` further waits for the VM's SSH server to become available via SSH, when this is the case VM's flag file will be removed
12) `vmPrologue.sh` creates the virtual nodes file

Guest OS boot sequence

The boot sequence is in general the same for standard Linux guests and OSv guests. However they differ in the technical details of how the several steps are implemented and realized

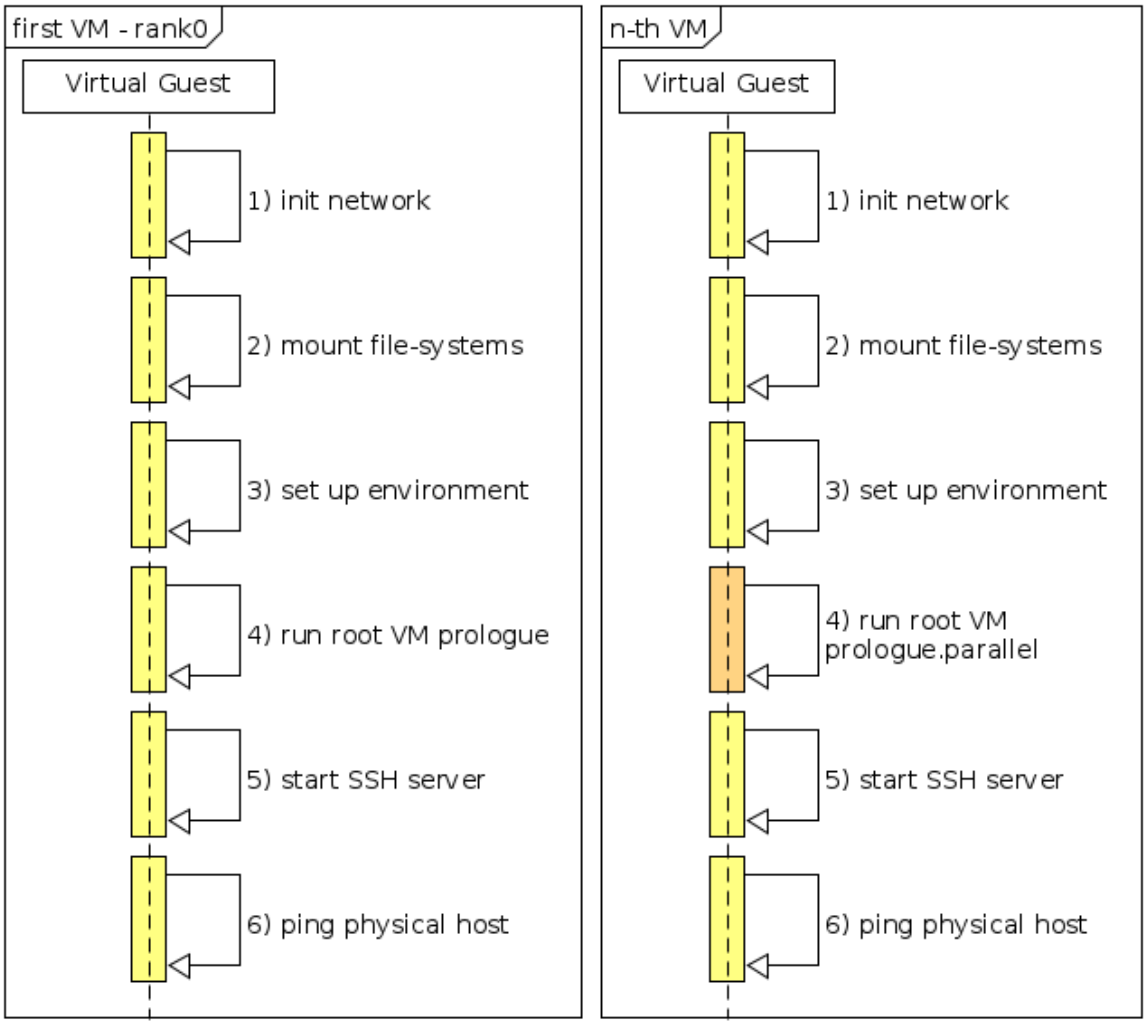

Figure 11. Guest's Boot Sequence.

1) The network is initialised: IP is fetched from DHCP server and NTP client
2) Shared file-systems are being mounted to `/home` and `/workspace`, optional disk is mounted in the first VM if available
3) Environment files need to be placed in `/etc/profile.d/` and Torque's job related files need to made available, too
4) Script root VM prologue is executed on the first node and on all other nodes the root VM prologue.parallel script is run
5) SSH server is started after the prologue(.parallel) scripts is run

6) Physical node is pinged to make the VM visible (to `apr -an`)

Job Script execution: environment preparations and job execution in VM

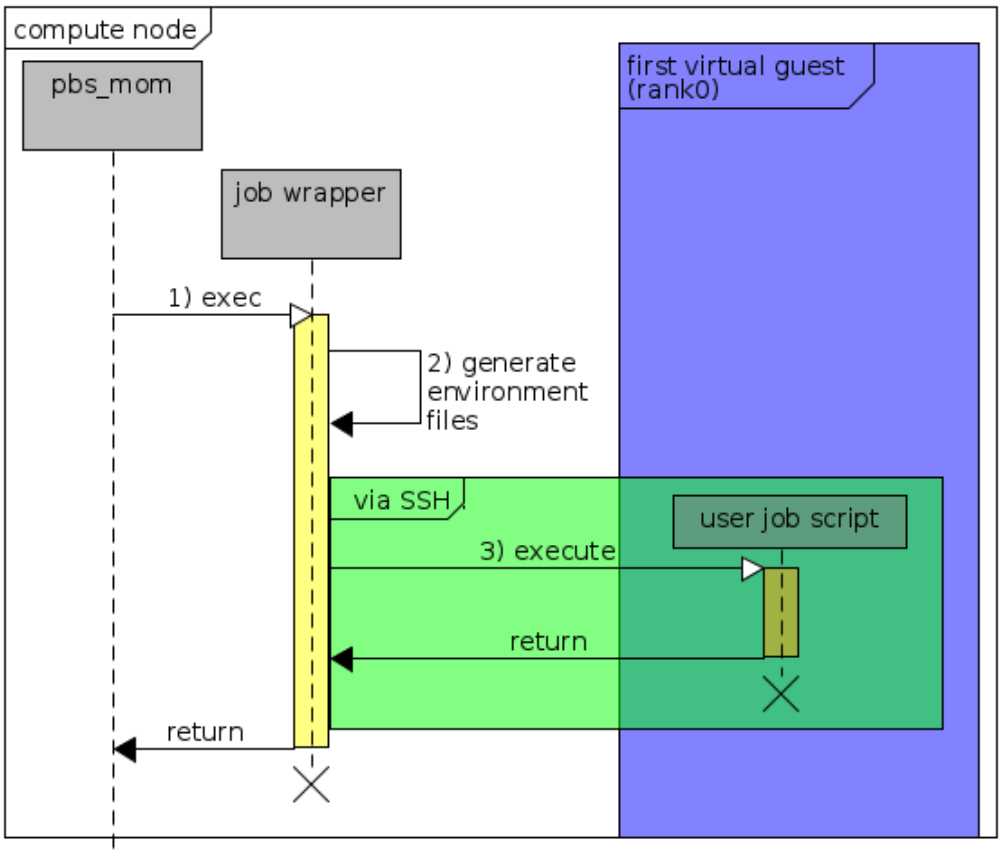

Figure 12. Job Execution Sequence.

After the prologue sequence is completed, the job execution phase starts. The wrapper script file `jobWrapper.sh` must be renamed to the user's job script name, in order to have the job's STDOUT/STDERR files named correctly. The naming schema for these files, if not defined by the user, is '`<scriptName>.[e|o]<jobID>`', i.e. '`myJob.sh.e177824`'.

1) The generated script `jobWrapper.sh` creates the job environment files for all guests
2) It then runs submitted user prologue script (if present)
3) It then runs submitted user vm prologue script (if present)
4) As next, the actual user job script is run in the first virtual guest of the job's resource allocation.
5) It then runs submitted user vm epilogue script (if present)

6) It then runs submitted user epilogue script (if present)

Epilogue sequence: VM Shutdown and Node cleanup

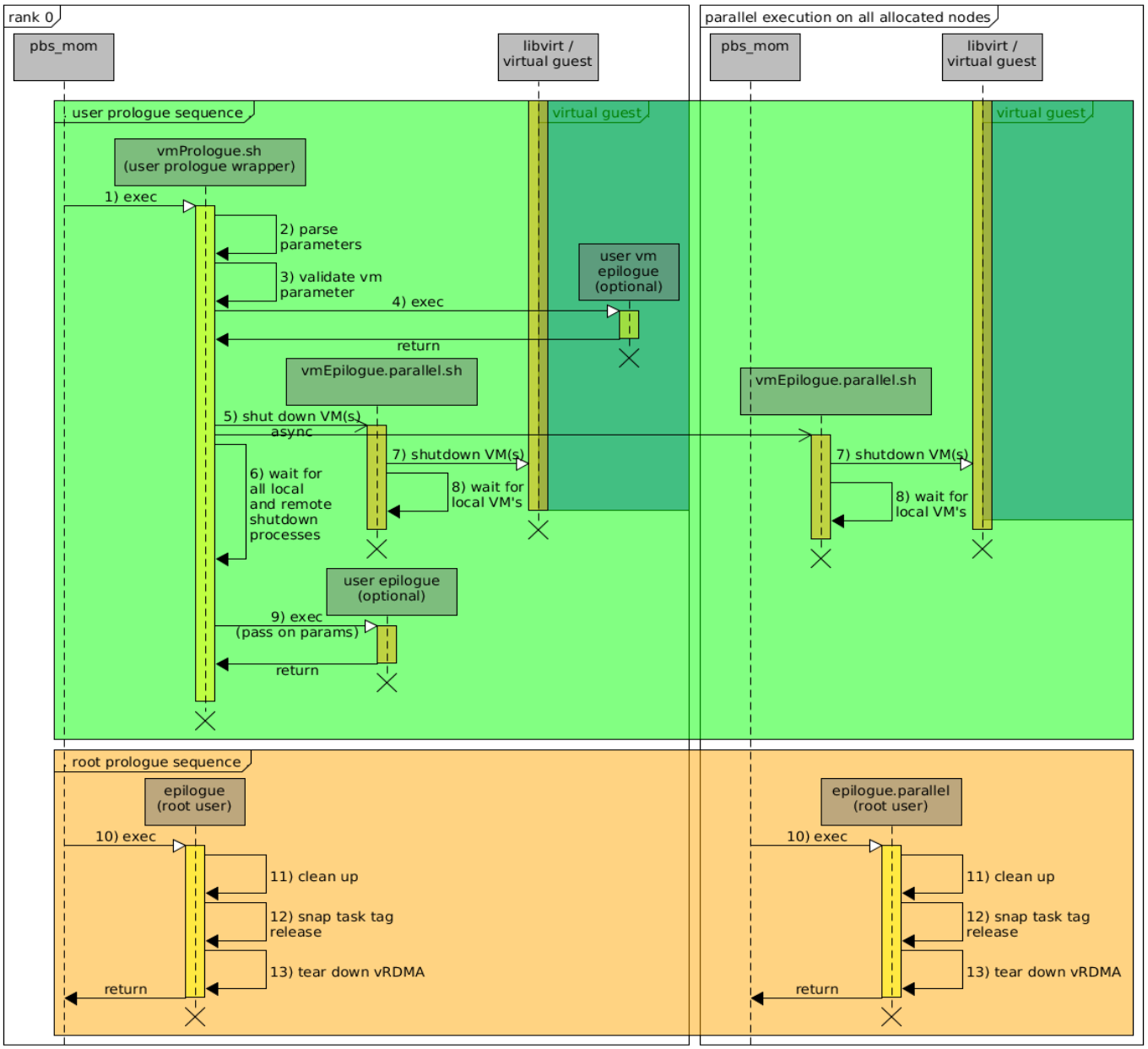

Figure 13. Extended Epilogue Sequence.

When the job is finished, the next sequence, epilogue scripts are executed.

1) The script `vmEpilogue.sh` parses parameters provided by Torque (job id, username, etc).

2) It then then cleans up all nodes by running script `vmEpilogue.parallel.sh` via SSH on all physical nodes allocated for user's job.
3) Script `vmEpilogue.parallel.sh` cleans up all VMs on the local host
    a) Release VM IPs
    b) If there is a persistent, optional user disk present in the first VM (rank0), it ensures data isn't lost
4) Script `vmEpilogue.sh` waits until all remote processes (`vmEpilogue.parallel.sh`) are finished / flag files are gone
5) Root `epilogue` and `epilogue.parallel` scripts unmount RAMdisk or cleans the shared file-system dir.
6) Root `epilogue` and `epilogue.parallel` release tag from snap monitoring tasks
7) Root `epilogue` and `epilogue.parallel` tears down vRDMA if enabled

Guest OS shutdown sequence

The shutdown sequence for the guests is the same for both standard Linux guests and OSv guests. However the functionality may be achieved in a slightly different way.

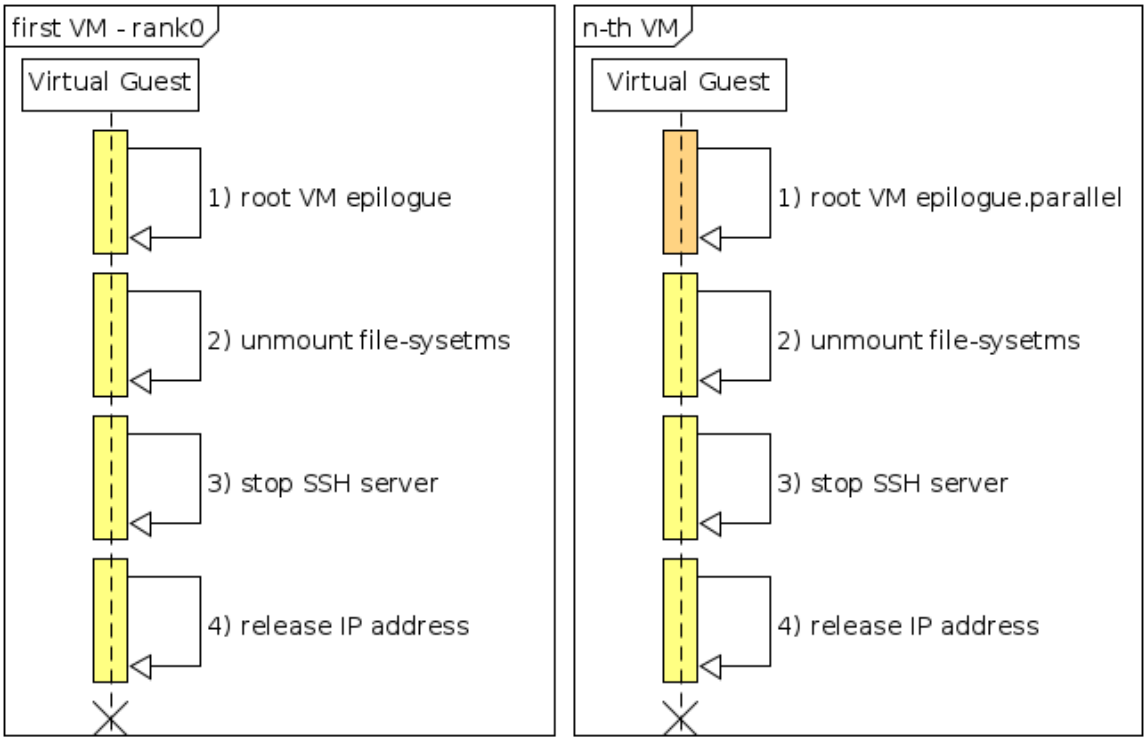

Figure 14. Guest's shutdown sequence.

Execution of the scripts, etc are triggered by the shutdown. After the illustrated steps the guest is stopped.

1) The root VM epilogue script is run on the first node and all others the root VM epilogue.parallel script.
2) File-systems and optional user disk that is mounted to the first node are unmounted.
3) SSH server is stopped
4) IP address is released

4.4.1.9 Considerations for Applications / Images

Application developers are requested to make use of the directory path '`/workspace`' to write out intermediate application data that benefit from fast shared file-systems. This path is not intended to keep any data beyond a job's runtime. It is mapped to the actual cluster's shared workspace file-system, configured by the cluster administrators, and may be wiped after a job has completed.

The directory path '`/home`' is also external storage and should be considered as slower mid term storage where data can reside after a job.

Optional, persistent, mid-term storage may be available under path '`/data`', in case there is a default disk defined globally or the user explicitly defined it at submission time. It will be mounted to the first virtual guest of a job's resource allocation where the user's job script is executed.

4.4.1.10 Limitations

There are five limitations identified to date; the first one is related to Open MPI. When a PBS environment is defined, the `mpirun` command expects a Torque `pbs_mom` compute node daemon running on all allocated (virtual) nodes it finds in the node file. This is circumvented by unsetting all PBS environment variables for the actual `mpirun` call, by the help of a wrapper script that is deployed via cloud-init metadata targets during the boot process of virtual guests. This prevents Open MPI to load the Torque module causing troubles with the virtual guests.

The second limitation is that interactive jobs in VMs are currently not possible from the technical point of view. This is due to Torque's behaviour of not allowing the submission of a job script, in our case the job wrapper, in combination with an interactive job. However, since the `jobWrapper.sh` script is required for preparing the VM's environment and other things that cannot happen before the actual job is run, extending Torque's source code by patching is required to implement functionality that accepts interactive jobs in combination with job scripts.

The third limitation is the missing support for accelerators and GPUs in virtual guests. Administrators can modify the domain XML template that is part of the MIKELANGELO software-stack and add a passthrough for any PCIe device in order to make it available to virtual guests. However, this would require that all nodes are equal in terms of the required domain XML configuration for the PCIe port passthrough.

Another limitation is that all guests are equal in terms of their properties. One cannot request e.g. four nodes with two VMs and eight nodes with one VM and twice as many VCPUs as the first four. Torque offers this feature for bare metal nodes.

The last limitation is that standard users cannot run jobs in virtual environments per default, since for booting any virtual guest, they need to be added to groups with appropriate rights, usually the libvirtd/kvm group. It has the disadvantage to enable users to boot *any* virtual guests on their own. There is currently not way to prevent it, and this could lead to problems with users booting images not approved by the infrastructure owners. Examples for undesired situations: IP may be not released afterwards, user root access in VM that mounts NFS shares, wrong uid for NFS access.

Future work will address these aforementioned limitations.

4.4.2 Torque Extensions for Standard Linux Guests

In this section we highlight the details of parts in the framework related to standard Linux guests, only. They are based on the general functionality and differ in some aspects from the artifacts related to OSv.

4.4.2.1 System Image Requirements

Standard Linux guest images intended to be used with the MIKELANGELO HPC software stack are standard cloud-images of up-to-date Debian or RedHat and their clones/derivatives. Required packages are:

`cloud-init`
: Required for customization (contextualisation) of virtual guests.

`systemd`
: Systemd is used to run the vm root user prologue and epilogue. Details follow in subsection *Script Execution* below.

4.4.2.2 Customization of Virtual Standard Guests

The customization of standard Linux guests is achieved with cloud mechanisms, namely the `cloud-init` package. The modules used are listed in the table below.

Table 2. Specific cloud-init commands used for customization of virtual guests.

Module name	Command used	Reason
bootcmd	groupadd useradd usermod mkdir unmount rmdir ln modprobe	Make sure the correct user is setup inside the vm
packages	install	Make sure the global packages needed are in place for the VM, like nfs-common
mounts	mount	This is the point where the external file systems get mounted.
manage-resolv-conf	n/a	DNS
write_files	n/a	Writes several files that are required inside the guest, for loading the job environment, NTP, defines custom services etc
runcmd	reconfigure openssh-server, ping, systemctl, start, pbs-vm-prologue.service systemctl, start, pbs-vm-prologue.service	Executes several actions at the end of the boot process required to make the guest available to the physical host, and for execution of root vm prologue and epilogue scripts
output	{all: '| tee -a /var/log/cloud-init-output.log'}	Collects all output from cloud-init in a predefined file. Actual path for the cloud-init log may differ for the various operating systems available.
final_message	n/a	Prints a final message at the end of cloud-init, containing the duration for the whole boot process and guest preparation.

4.4.2.3 Preparation of Virtual Job Environment

The `jobWrapper.sh` script uses SSH to connect to the virtual guest and then sources the additional VM PBS environment variables that may be expected by job scripts, then it executes the user's job script.

The nodefile that is used inside the VM is mounted as a read-only passthrough device with VirtFS 9p-virtio [35, 36].

4.4.2.4 Script Execution

Root user VM prologue and epilogue scripts are executed with systemd at the point in time of the boot process where cloud-init is done and all services are started, except for the SSH server.

4.4.2.5 Virtual Node Access

As HPC users are used to being able to connect to nodes hosting their jobs, connectivity to virtual nodes hosting their jobs is also supported.

4.4.3 Torque Extensions for OSv

The extensions required for OSv are almost in line with standard Linux guests, however there are some minor differences in the way OSv images are handled, e.g. in regards to metadata and interactive jobs. OSv furthermore requires modifications of the MPI startup, due to its nature of being a single-process container, while MPI depends on multiple processes for its polling, running its daemons, etc.

Instead of relying on complex integration logic, we have decided that the underlying components required for integration of HPC applications with OSv should resemble the existing interfaces as much as possible. The rationale for this decision is the fact that the MIKELANGELO project is only going to integrate with a single HPC platform (Torque), but the results of the project are suitable for alternative technology stacks (for example the Slurm workload manager [37]). Supporting well known interfaces is going to broaden the exploitation potential.

The most notable requirements for HPC integration are presented next.

1. **Cloud-init**
   While we already explained the requirements for OSv's cloud-init module in the previous section, an additional requirement for HPC is mandatory. Torque on its own does not provide a dedicated cloud-init service that instances would be querying for meta- and user-data. HPC integration thus relies on the *No cloud* [23] mechanism by

attaching a secondary disk device to a VM. OSv has already provided a similar solution for passing a single configuration file to the instance. HPC integration must ensure that a properly formatted file is created and attached to OSv-based instances.

2. **Execution of MPI applications**
   To start an MPI application in Linux, the command is submitted over a remote shell connection (SSH) as if the user has logged into the machine and executed the command manually, but without a (real) terminal session. OSv does not provide a remote shell, however it provides a RESTful API with similar functionalities. The HPC integration must ensure that the corresponding command is passed to the aforementioned API instead of a remote shell.

Preliminary integration demonstrating a proof of concept for running OSv instances through HPC infrastructure has already been done in collaboration between work packages 4 (guest operating system and application packaging), 5 (HPC integration) and 6 (use cases, in particular cancellous bones and aerodynamics).

4.4.3.1 Image Requirements

Required packages for building OSv-based images to be executed on an HPC infrastructure are already provided in WP4: OSv kernel base, Open MPI, cloud-init and HTTP server. Additional packages from a pre-built repository, for example a simple command line interface for debugging the image, can be added to the image. Application specific binaries, required libraries and other files may be copied onto the target image.

4.4.3.2 Customization of Virtual OSv Guests

The most important difference between OSv and Linux images is that OSv images have to be prebuilt. MIKELANGELO Package Manager facilitates the preparation of these images in advance, along with capabilities for testing locally. The cloud-init module in OSv does not provide any support for installation of additional packages. It does, however, allow for customisation of specific files and uploading new ones (for example, to reconfigure the application specifically for the use case).

4.4.3.3 Preparation of Virtual Job Environment

Instead of relying on a remote SSH connection to OSv instances, its HTTP(S) API should be used. We have already demonstrated the use of this API for executing MPI applications by providing a PLM (Process Launch Module) component to Open MPI. The same API can be used to configure environment variables required by Torque and its corresponding daemons.

4.4.3.4 Script Execution

OSv does not support execution of scripts. Therefore, any configuration should either be done prior to the image being built, or through the HTTP API. Alternatively, the HPC integration might also consider deployment of an additional Linux-based VM that users get access to. Because such a VM would be deployed for the same job, it would have access to the same configuration and data files and would consequently allow users to address their specific requirements for the execution of jobs.

4.4.3.5 Virtual Node Access

Remote access to OSv instances is not possible at the moment. OSv does provide a command line interface with limited capabilities which would not suffice for typical HPC users. We are going to investigate this further in the remaining phases of the project.

4.4.3.6 Execution of Integrated Components

The component functionalities defined for a job are applied to the physical node in the root level prologue script, before the job starts. It is removed in the root user level epilogue, after a job has finished. The commands for setting it up and tearing it down require execution with root permissions, thus they are executed in the root level scripts.

4.4.4 IOcm

The integration of the IO core manager (IOcm), in order to increase I/O operations for virtual guests, into our extensions for Torque [30], consists on the one hand of configuration, command line and inline parameters, and on the other hand it requires a certain kernel in place, which limits its availability.

At this point in time the kernel source code is located inside the MIKELANGELO GitHub repository [38]. The patches needed for the kernel are for the 3.18 kernel and will be upstreamed by IBM. To build the kernel, the basic Ubuntu 14.04 kernel config was used, to be as compatible with the operating system as possible.

The first part are three additional parameters that the user can define on the job submission command line or inline in the job script's header section. These additional (optional) qsub command line parameters for IOcm are prefixed with '`-vm`':

```
iocm=true|false
iocm_min_cores=<number gt 0>
iocm_max_cores=<number gt 0 and ge iocm_min_cores>
```

The second part of the integration are the global admin configuration parameters. The corresponding global admin configuration parameters, limit the user choices in terms of

IOcm enabled at all, as well as boundary values for the min and max core count an user can define. Global admin configuration cannot be be overridden by the user parameters. The global admin configuration parameters for IOcm are defined in file `config.sh`.

```
IOCM_ENABLED=true|false
IOCM_MIN_CPUS=<number gt 0>
IOCM_MAX_CPUS=<number gt 0 and ge iocm_min_cores>
IOCM_NODES=<regex for hostnames, i.e. [*] >
IOCM_SCRIPT_DIR=<iocm installation dir>
```

Furthermore, there are default parameters defined in the global admin configuration that come into place when `IOCM_ENABLED` is globally set to true and the user does not provide any `IOCM_*` parameters:

```
IOCM_ENABLED_DEFAULT=true|false
IOCM_MIN_CPUS_DEFAULT=<number gt 0>
IOCM_MAX_CPUS_DEFAULT=true|false
```

4.4.5 *Virtual RDMA*

Integration of vRDMA connectivity to enable multiple virtual guests to share the Infiniband network of the underlying host system into our extensions for Torque [30], consists on the one hand of configuration, command line and inline parameters, and on the other hand it requires a customized version of QEMU in place, which limits its updatability by the operating system's package management system (`i.e. dpkg, yum, zypper`, etc).

In D4.1 [5], we have presented example scripts for starting up and stopping the virtual RDMA environment in the Appendix section. These scripts were prepared to be run by a user with root privilege, subsequently normal users could use the setup created by the scripts.

The first part of the integration is one additional parameter that the user can define on the job submission command line or inline in the job script's header section. The additional (optional) qsub command line parameters for vRDMA is prefixed with '`-vm`':

```
vrdma=true|false
```

The second part of the integration are the global admin configuration parameters. The corresponding global admin configuration parameters define if vRDMA is enabled at all and on which hosts capable hardware is available. Global admin configuration cannot be be overridden by the user parameter. The global admin configuration parameters for vRDMA are defined in file `config.sh`.

```
VRDMA_ENABLED=true|false
VRDMA_NODES=<regex for hostnames, i.e. [*] >
VRDMA_SCRIPT_DIR=<vRDMA scripts installation dir>
```

Furthermore, there is a default parameter defined in the global admin configuration that applies when `VRDMA_ENABLED` is globally set to true and the user does not provide it:

```
VRDMA_ENABLED_DEFAULT=true|false
```

The startup script will setup the environment variables when a job request is initiated by Torque, then Libvirt will be able to access the databases and management sockets of Open vSwitch for the virtual machines.

For starting virtual machines on virtual RDMA nodes, a special libvirt configuration has to be used. As described in the Appendix of D4.1 [5], the created vhost-user port has to be specified as a network device in the Libvirt configuration file for the guest OS. Please note that, on each node, two vhost-user ports are created on the same Open vSwitch bridge, which combines these two vhost-user ports into one InfiniBand physical port. This allows two guests on the same host to use different vhost-user ports at the same time respectively, and thus to use shared memory protocol for inter-host communication. This setting is also prepared to run several guests on the same node. Templates for the correct settings have been provided in the M18 release.

An example interface configuration for Libvirt's domain XML format is shown as follows:

```
<interface type='vhostuser'>
  <mac address='mac_address_for_guest'/>
  <source
     type='unix'
     path='path_to_corresponding_OVS_DB_socket'
     mode='client'
  />
  <model type='virtio'/>
  <address
     type='pci'
     domain='0x0000'
     bus='0x00'
     slot='0x03'
     function='0x0'
   />
</interface>
```

An additional DHCP server is created on each virtual RDMA host when the startup script is called by Torque. The IP range managed by the DHCP server is defined in the file `config.sh` and serves dynamic IP addresses for the local Open vSwitch bridge and the virtual guests. The DHCP configuration file is at the default path of the node (`/etc/dhcp/dhcpd.conf`), As all the virtual RDMA nodes are connected with a local private InfiniBand network, the

Open vSwitch bridges and the guests will be able to broadcast the DHCP requests and acknowledgements to each other, so they will not have conflicts of IP address assignment.

4.4.6 *Snap*

Intel's monitoring component, called snap, allows the detailed tracking of the resource usage of batch jobs on a per user and per job basis, covering many metrics that are defined in selected plug-ins. The association of resource allocations and consumption is achieved by the help of global monitoring-tasks that are tagged with the user's name and the job's id. Each of these monitoring tasks is dedicated to a certain compute node.

Monitoring of resources runs continuously in the HPC environment and is not started/stopped on a per-job basis. The monitoring tasks are always started during the boot of physical nodes and stopped in the shut down. Whenever a job starts, the monitoring task will be tagged with the job's ID and the user's name to identify the resources the job consumes.

Snap tag format used

```
experiment:experiment:nr, job_number: $JOBID
```

Example tag string, for job ID '2273.vsbase2'

```
experiment:hpc:nr, job_number: 2273.vsbase2
```

Tagging of tasks takes places at the earliest point in time possible in a job's lifecycle. The root prologue script is executed as root user before any user prologue scripts or the actual job script is run. The tag is released from the task at the end of a job's root epilogue, the latest point in time in a job's life cycle where it is possible. This way we ensure capturing almost all the resources a job consumes, except for some cycles that are used for job submission, job scheduling and job deployment.

For the integration of snap there are the global admin configuration parameters, only. The global admin configuration parameters for IOcm are defined in file `config.sh`.

```
SNAP_MONITORING_ENABLED=true|false
SNAP_SCRIPT_DIR=<path to the snap installation>
```

For the continuous global measurements the following metrics for snap are available and can be defined in file `snap-common.sh`. Please refer to Appendix A.2 for the details about available metrics.

The collected metrics are published to an InfluxDB database server on which grafana web-dashboards can be viewed with a browser.

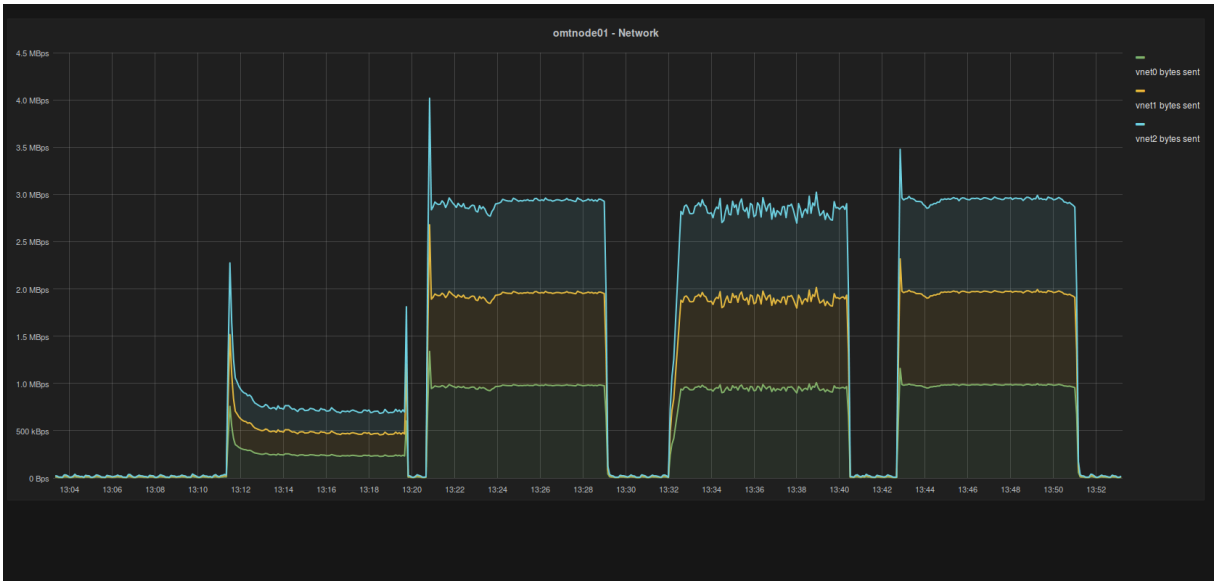

Figure 15. Visualization of snap metrics with grafana [39].

In the screenshot above, network traffic for 3 different VMs, communicating with each other, is illustrated.

4.4.7 *SCAM and Torque NUMA Nodes*

The security module is not needed for the HPC deployment of the MIKELANGELO software stack. The reason is simple, there are no shared caches or memory, since all nodes in HPC environments are per default allocated exclusively for a certain job. Even when using the NUMA node support within Torque where multiple users may share a physical node, there is no shared cache or memory that could be attacked [40].

4.4.8 *OSv Support for HPC Workloads*

An HPC workload typically consists of the main application, a set of supporting libraries and a shared storage space. It is executed in an environment already provisioned for a single user and does not require specific hardware, apart from that used for efficient communication with other workers. While virtualisation allows for the transparent use of resources regardless of whether the workload is running on physical nodes or virtualized, it does bring additional overhead, such as the need for large images, increased boot times and redundant security mechanisms in the guest OS. The unikernel approach on the other hand drastically reduces all of these overheads compared to standard Linux guests.. The size of the kernel may be just a few megabytes, allowing sub-second boot times, and typical security measures are irrelevant due to its single-user address space.

In the context of High Performance Computing, providing support for efficient parallelisation using MPI is one of the most significant requirements. Report D5.4 [41] (First report on the

Integration of sKVM and OSv with HPC) has already presented detailed analysis of Open MPI and in particular its execution model. The description introduced the main components involved, namely the `mpirun` application, `orted` daemon and SSH (Secure Shell) communication channel. The `mpirun` application is the main entry point of any MPI-enabled application, such as OpenFOAM or the MIKELANGELO Cancellous bones application. It provides a common middleware responsible for bootstrapping the parallel execution on a multitude of distributed hosts. The `mpirun` binary connects to the given hosts using SSH and launches the daemon that calls back to the master process. Once all daemons are registered, they receive the actual workload (MPI-enabled application), properly configured for parallel execution.

During the execution `mpirun` monitors the status of all worker processes. Failure of a single process typically results in a failure of the entire workload. Furthermore, the mpirun process collects the data from worker processes and aggregates any log output that the application might be producing.

The aforementioned report also presented several caveats of the OSv unikernel from the perspective of MPI applications as well as some preliminary changes the MIKELANGELO project already contributed. The most important new functionality was the support for running multiple instances of the same application binary in a single OSv instance. This allowed the integration work package WP6 to modify the `orted` daemon to launch the requested number of threads instead of forking new processes. However, this change was still limited to a single OSv instance preventing stable execution on multiple physical hosts.

Most of the research and development related to MPI in the last six months was therefore allocated to resolving this issue. Instead of working with a wrapper (a proxy) to communicate with the underlying OSv instances, we have decided to completely replace the way MPI instantiates worker processes. Default configuration uses an SSH based PLM (process launch module) to start orted daemon on remote nodes. We implemented an additional PLM module named `osvrest` which uses the OSv HTTP REST API to start new Open MPI worker processes. The `osvrest` PLM module requires that OSv VMs are started before running the mpirun command which is in line with the current status of the HPC integration where the virtualisation module for Torque ensures virtual machine instances are instantiated on allocated hosts. The fact that these changes are provided by a separate PLM component ensures that the deployed Open MPI can be used to execute on physical hosts and Linux or OSv virtual machines. This allows end users to choose their runtime environment when submitting jobs to the workload manager.

The initial version required that the master process (the one running only `mpirun`) is executed separately from the other processes which is significantly different from the typical

workflow. Usually, the `mpirun` is executed on one of the nodes allocated for the execution, along with a designated number of worker processes.

This cumbersome requirement that a dedicated VM, even if very low on resources, has to be launched in addition to all other VMs doing the actual work, also imposes a problem for the integration with the HPC batch system (Torque). It would require a different deployment strategy than in case of Linux virtual machines (for every MPI job, one more VM has to be launched specifically for the master process). Consequently, we have further analysed the limitations of the existing implementation and resolved all of the issues allowing us to launch MPI based applications exactly as they are in the case of a Linux operating system.

Another high level requirement that has already been addressed in MIKELANGELO is support for NFS. HPC applications rarely store data locally. Instead, worker processes access data from shared storage accessible through NFS or Lustre [16]. Because OSv already implements an abstract file system (VFS; Virtual File System), it was possible to integrate the NFS using the freely available libnfs library [42]. However, the performance evaluation revealed that this implementation is slower compared to the one used in Linux. To address this issue, additional analysis will be performed to understand and improve the performance.

The HPC integration uses the cloud-init mechanism to contextualise workers running in virtual machines (Linux). Cloud-init is a standard mechanism used in a variety of cloud providers and cloud middlewares, and most standard Linux distributions already have required modules enabled in their images used for cloud distribution. Contrary to cloud environments, HPC does not provide a cloud-init metadata service and, in order to reduce the number of external dependencies for the deployment, current HPC contextualisation relies on the No cloud [28] datasource delivery mechanism. In this case the additional data is passed into the target VM as a secondary disk volume.

OSv already offers the cloud-init module. Only Amazon's AWS and Google's GCE data sources were supported initially. The module was therefore extended with a mechanism similar to *No cloud* where cloud-init data is read from a secondary disk device.

The cloud-init implementation supports numerous configuration options and HPC integration uses several of them in the case of Linux-based virtual machines: setting the hostname and the user, requesting packages to be installed, specifying the shared storage mount points and running arbitrary commands, to name a few. Most of these are not available nor applicable in OSv, but those that are relevant are:

1. **Setting the hostname**

By default, each OSv instance sets hostname to *osv.local*. In order to be able to distinguish between running instances, a patch for cloud-init has already been created (not upstreamed yet).

2. **Specifying shared storage**
   This functionality is not supported yet. It is requested that arbitrary numbers of additional block devices (either remote or local) can be attached to the running OSv instance. Currently, remote devices should be mounted as NFS shares. Local devices should be available through corresponding libvirt domain specification.
3. **Running commands**
   Arbitrary shell commands can be executed with this configuration option in Linux environments. Although this is not suitable in an unikernel, it is nevertheless important that required workloads can be started using cloud-init instead of having this information stored in the image itself.

4.5 Requirements and Future work

Several new requirements have been identified during the HPC and Cloud integration cycles, most of them are already addressed or are currently in the process of being implemented. Some requirements are essential for the integration of all components described in the previous sections into Cloud and HPC environments. In the next subsection each component and its associated requirements for both Cloud and HPC integration is highlighted.

4.5.1 Cloud

The future work of the cloud integration consists of two separate parts. The first part regards the development of new features for individual components. The second part regards the integration of said components in a coherent framework. The latter includes the implementation of cross-layer optimization for the cloud. Future work on individual components and cross-layer optimization plans have been described in detail already in Section 4.2 and 4.3. Here we provide a short summary of the future work.

IOcm will be extended by providing a local resource manager for the assignment of CPU resources to VMs. Virtual RDMA will incorporate the use of normal NICs in the cloud environment. In addition, the integration of virtual RDMA with the OpenStack deployment will be implemented. SCAM will be ported to GWDG's hardware and embedded as part of the OpenStack testbed. OSv will be extended by further capabilities in cloud-init, such as *phone-home* functionality. Snap will be optimized for data transfer, reduced redundancy, and improved cloud integration. OSv will be extended to work in conjunction with Capstan to provide additional flexibility when deploying applications with MPM.

Cross-layer optimization will be tackled from the ground up as future work. The first steps include the deployment of snap and pooling of data as a basis for optimization. Then experiments will be conducted to optimize the use of IO resources. In the same vein an online scheduler for OpenStack will be created to facilitate the movement of VMs for optimization.

4.5.2 HPC

Future work on the HPC software-stack focuses on the one hand on the limitations in place, and on the other hand on new functionality to be implemented in wrapper scripts and Torque.

Limitations to be addressed:

1) Users currently need to be able to boot virtual guests, since VMs are booted in the wrapper for Torque's user prologue. It is desired to boot VMs as root user only, to get rid of the security impact that comes along when users can choose the uid for NFS access and break the data privacy of other users.
2) Interactive Jobs with virtual guests require patching of Torque's source code. We already have a first patch, however it needs further investigation as the console is no longer responsive after a job has completed.
3) Missing support for accelerators and GPUs. We will investigate possibilities to support GPUs with virtualization capabilities as well as support for different domain XML templates to cover generic types of accelerators.
4) Allow mixed resource requests for -vm parameters, too, like it is possible to request from Torque, i.e. two nodes with one core and three nodes with eight cores combined in one job submission.

Functional features that are targeted as future work during the next 12 months as unsorted list:

- Finalize OSv HPC integration
- Checkpoint+Restart functionality for virtual guests
- Support GPU and generic PCIe accelerator cards (at least a dynamic passthrough)
- Node health monitoring with snap
- Live migration of running guests under load to spare nodes
- Spare node management within Torque (requires a patch for Torque)
- Auto-migration of running guests to spare nodes if node health monitoring reports degrading hardware

- Isolated job-exclusive private networks with OpenFlow/Open vSwitch
- Enable Torque to execute interactive jobs in virtual guests (requires a patch for Torque)
- Get rid of required libvirtd user group requirement for users (boot VMs as root user)

Investigation on possibilities and functionality is targeted for:

1) Overcommitting cores, useful for io- or memory-bound HPC applications
2) Detection of optimal alignment for the vcpu-pinning
3) MPM capabilities, with the focus on dynamically building OSv images for submitted jobs, before they start.
4) Automatic detection of and support for vRDMA and IOcm capable nodes within Torque.

4.6 Key Takeaways / Concluding Remarks

During the last reporting period the first progress towards an unified MIKELANGELO HPC/Cloud software stack had been achieved. Several performance improvements, especially for virtualized I/O, came into place. Further, the usability for OSv's application packing process has been improved and many functionalities throughout the whole MIKELANGELO software-stack was implemented.

Major progress in regard to the integration of sKVM and monitoring components developed within the project into the Cloud and HPC software-stacks and infrastructures, validated by the use cases described in the Use Deliverable D2.2 [42]. Many technical challenges were resolved during the implementation and component integration cycles and a few limitation had been identified, leading over to new requirements. Some of these new requirements are already in process to be addressed and resolved.

5 Concluding Remarks

This deliverable presented the current status of the MIKELANGELO project. Individual components have been reviewed first, followed by the two approaches to the integration of these components, namely cloud and virtualised HPC.

The first important conclusion of this deliverable is that contrary to our results in M12, where the main focus was on underlying components, M18 already offers partially integrated environments. Even more importantly, initial architecture designs have proven to be mostly valid for our integration. Even this early integrated prototype released as part of the milestone MS3 already supports execution of complex scientific applications, such as Open MPI and OpenFOAM, and big data applications, like Hadoop HDFS, on top of the modified hypervisor with a lightweight unikernel OSv. Users are furthermore able to run their workloads and applications in commonly used middlewares (Torque and OpenStack) either traditionally or exploiting MIKELANGELO's enhancements.

In collaboration with work package 6, initial evaluations have been conducted and reported in deliverable D6.1. Implementation, integration and evaluation has lead to new requirements, also presented in this report. Requirements were gathered both from the perspective of use cases as well as from technical perspective. Former requirements are more high-level because allowing use cases to provide only the final goal, which should be as generic as possible to support external use cases as well. Technical requirements have been designed primarily based on the work done so far and the limitations we have identified. Several requirements were introduced after consulting with external open source communities (most notably, IOcm configuration interface).

We will continue to periodically review all architectural aspects of the MIKELANGELO project as the work continues towards two other prototypes released in M24 and M30. Particular focus will be dedicated to assessment of merging of the two architectures. All changes will be reflected in the final report on architecture in M30, just before the final 6 months of refinements and improvements of all components of the project.

6 References and Applicable Documents

[1] The MIKELANGELO project, http://www.mikelangelo-project.eu/

[2] MIKELANGELO D2.16 - The First OSv Guest Operating System MIKELANGELO Architecture, http://www.mikelangelo-project.eu/deliverable-d2-16/

[3] MIKELANGELO D2.13 - The first sKVM hypervisor architecture, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-13/

[4] MIKELANGELO D3.1 - The First Super KVM – Fast virtual I/O hypervisor, https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP3.1-IBM-v1.0.pdf

[5] MIKELANGELO D4.1 - The First Report on I/O Aspects, https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP4.1-Huawei-DE_v2.0.pdf

[6] MIKELANGELO D6.1 - First report on the Architecture and Implementation Evaluation, http://www.mikelangelo-project.eu/deliverables/deliverable-d6-1/

[7] Mellanox. (2015). RDMA Aware Networks Programming User Manual, 1–216. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, Retrieved from www.mellanox.com

[8] Valgrind Project Page, http://valgrind.org/

[9] Calgrind – Valgrind User Manual, http://valgrind.org/docs/manual/cl-manual.html

[10] MIKELANGELO D2.10, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-10/

[11] Hyv - A hybrid I/O virtualization framework, https://www.research.ibm.com/labs/zurich/sto/security/zac_hyv.html

[12] Hyv on GitHub, https://github.com/zrlio/hyv

[13] MIKELANGELO D3.4 - sKVM Security Concepts – First Version, https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP3.4-BGU-v1.0.pdf

[14] MIKELANGELO D4.4 - OSv – Guest Operating System – First Version, https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP4.4-ScyllaDB-V1.0.pdf

[15] ZFS on Linux, http://zfsonlinux.org/

[16] Lustre Project Page, http://lustre.org/

[17] Seastar Documentation, http://docs.seastar-project.org/master/index.html

[18] Capstan, a tool for packaging and running your application on OSv, https://github.com/cloudius-systems/capstan

[19] Unik, https://github.com/emc-advanced-dev/unik

[20] Open MPI Project Page, https://www.open-mpi.org/

[21] OpenFOAM Project Page, http://www.openfoam.com/

[22] Java, https://www.java.com/en/

[23] Cloud-init http://cloudinit.readthedocs.io

[24] Amazon EC2, https://aws.amazon.com/ec2/

[25] Cloud-init Documentation, http://cloudinit.readthedocs.io/en/latest/index.html

[26] Cloud-init Datasources, http://cloudinit.readthedocs.io/en/latest/topics/datasources.html#ec2

[27] Google Compute Engine, https://cloud.google.com/compute/

[28] Cloudn-init / NoCloud tar, http://cloudinit.readthedocs.io/en/latest/topics/datasources.html#no-cloud

[29] HadoopFS User Guide, https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

[30] Torque PBS, http://www.adaptivecomputing.com/products/open-source/torque/

[31] Snap, https://github.com/intelsdi-x/snap

[32] http://trustedanalytics.org/

[33] OpenStack Murano, https://wiki.openstack.org/wiki/Murano

[34] Mellanox. (2015). RDMA Aware Networks Programming User Manual, 1–216. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, Retrieved from www.mellanox.com

[35] VirtFS, http://www.linux-kvm.org/page/VirtFS

[36] VirtFS 9p VirtIO, http://www.linux-kvm.org/page/9p_virtio

[37] SLURM, http://slurm.schedmd.com/

[38] MIKELANGELO GitHub repository, https://github.com/mikelangelo-project

[39] Grafana, http://grafana.org/

[40] Cisco Blog Post "Process and memory affinity: why do you care", http://blogs.cisco.com/performance/process-and-memory-affinity-why-do-you-care

[41] MIKELANGELO D5.4, https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP5.4-USTUTT-v1.0.pdf

[42] libnfs, https://github.com/sahlberg/libnfs

[43] MIKELANGELO D2.2, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-2/

[44] QEMU PC targets, http://qemu.weilnetz.de/qemu-doc.html#QEMU-PC-System-emulator

[45] QEMU non-PC targets, http://qemu.weilnetz.de/qemu-doc.html#QEMU-System-emulator-for-non-PC-targets

[46] Modules System, http://modules.sourceforge.net/

Appendix A. Technical Details on the HPC-Cloud Infrastructure

This appendix contains in first place technical details.

A.1 Virtual RDMA Set-Up and Tear-Down Commands

Table 3 shows several important variables that are configured by the startup script and provides short descriptions and default values for the HPC infrastructure at USTUTT.

Table 3. Several important environment variables that are configured by the startup script.

Variable Name	Default Value	Description
VRDMA_BRIDGE	vrdma-br	Name of the RDMA virtual bridge created by Open vSwitch
IB0	dpdk0	The configured InfiniBand port name in RoCE mode.
SOCKET0_MEM	2048	MB of memory attached to the corresponding socket that the RDMA device is connected to.
IB_PCI_ADDR	0000:05:00.0	PCI address of the RDMA device.
LIBVIRT_URI	qemu+unix:///system?socket=$LIBVIRT_RUN_DIR/libvirt-sock	Libvirt socket created by Libvirt daemon for specific compute node. This has to be correctly configured, otherwise virsh commands will not work.
DB_SOCK	$OVS_DIR/var/run/openvswitch/ovs-db-$NODENAME.sock	Socket for the main database of the Open vSwitch server. This has to be correctly configured, otherwise ovs commands will not work. This socket has to be be readable by the user.

When launching the startup script, it first sets all the necessary environment variables and necessary command aliases. Then hugepage support is enabled by creating a new device and mounting one GB of memory to the device. For a CPU that has direct support for 1GB hugepages, it can be directly mounted using the following command as user root in the script:

```
mount -t hugetlbfs -o pagesize=1G none /dev/hugepages
```

For a CPU that does not support it, we have to use 2 megabytes contiguous memory regions for each NUMA node. The following commands are used in the startup script:

```
root@host-vrdma:~#echo 8192 > \
/sys/devices/system/node/node0/hugepages/hugepages-
2048kB/nr_hugepages

root@host-vrdma:~# echo 8192 >\
sys/devices/system/node/node1/hugepages/hugepages-
2048kB/nr_hugepages
```

Open vSwitch server and daemon are started using the following command lines:

```
root@host-vrdma:~# ovsdb-tool create $OVS_DATABASE \
                     $OVS_SHARE_DIR/vswitch.ovsschema

root@host-vrdma:~# ovsdb-server $OVS_DATABASE \
          --remote=punix:$DB_SOCK \
          --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
                    --pidfile=$OVS_SERVER_PID --detach \
                    --log-file=$OVS_SERVER_LOG

root@host-vrdma:~# ovs-vsctl --db=unix:$DB_SOCK --no-wait init

root@host-vrdma:~#ovs-vswitchd --dpdk -c 0x1 -n 4 -w $IB_PCI_ADDR\
          --socket-mem $SOCKET0_MEM,$SOCKET1_MEM \
          --unix:$DB_SOCK --pidfile=$OVS_DAEMON_PID --detach \
          --log-file=$OVS_DAEMON_LOG
```

To create the virtual RDMA bridge using Open vSwitch, the following command is used:

```
root@host-vrdma:~#                ovs-vsctl                --no-wait\
          --db=unix:$DB_SOCK add-br $VRDMA_BRIDGE \
          --set bridge $VRDMA_BRIDGE datapath_type=netdev
```

For binding the physical InfiniBand port and creating a DPDK vhost-user port on the binding, the following commands are required:

```
root@host-vrdma:~#   ovs-vsctl   --db=unix:$DB_SOCK   --no-wait   \
     add-port $VRDMA_BRIDGE $IB0 --set Interface $IB0 type=dpdk

root@host-vrdma:~#   ovs-vsctl   --db=unix:$DB_SOCK   --no-wait   \
     add-port        $VRDMA_BRIDGE        $DPDKVHOST_IB0_0        \
     --set Interface $DPDKVHOST_IB0_0 type=dpdkvhostuser

root@host-vrdma:~#   ovs-vsctl   --db=unix:$DB_SOCK   --no-wait   \
     add-port        $VRDMA_BRIDGE        $DPDKVHOST_IB0_1        \
     --set Interface $DPDKVHOST_IB0_1 type=dpdkvhostuser
```

A.2 Snap Metrics

Snap supports the collection of many hundreds of metrics through dozens of plugins targeting various sources of telemetry across the hardware and software stack. The following list of metrics are currently being collected by snap from the integrated HPC stack in MIKELANGELO to facilitate initial performance analysis. The list will be updated as new functionality is integrated and new metrics become of interest.

```
/intel/psutil/load/load1
/intel/psutil/load/load15
/intel/psutil/load/load5
/intel/psutil/net/all/bytes_recv
/intel/psutil/net/all/bytes_sent
/intel/psutil/net/all/dropin
/intel/psutil/net/all/dropout
/intel/psutil/net/all/errin
/intel/psutil/net/all/errout
/intel/psutil/net/all/packets_recv
/intel/psutil/net/all/packets_sent
/intel/psutil/vm/active
/intel/psutil/vm/available
/intel/psutil/vm/buffers
/intel/psutil/vm/cached
/intel/psutil/vm/free
/intel/psutil/vm/inactive
/intel/psutil/vm/total
/intel/psutil/vm/used
/intel/psutil/vm/used_percent
/intel/psutil/vm/wired
/intel/Linux/iostat/avg-cpu/%user
/intel/Linux/iostat/avg-cpu/%nice
/intel/Linux/iostat/avg-cpu/%system
/intel/Linux/iostat/avg-cpu/%iowait
/intel/Linux/iostat/avg-cpu/%steal
/intel/Linux/iostat/avg-cpu/%idle
/intel/Linux/iostat/device/ALL/rrqm_per_sec
/intel/Linux/iostat/device/ALL/wrqm_per_sec
/intel/Linux/iostat/device/ALL/r_per_sec
/intel/Linux/iostat/device/ALL/w_per_sec
/intel/Linux/iostat/device/ALL/rkB_per_sec
/intel/Linux/iostat/device/ALL/wkB_per_sec
/intel/Linux/iostat/device/ALL/avgrq-sz
/intel/Linux/iostat/device/ALL/avgqu-sz
/intel/Linux/iostat/device/ALL/await
/intel/Linux/iostat/device/ALL/r_await
```

```
/intel/Linux/iostat/device/ALL/w_await
/intel/Linux/iostat/device/ALL/svctm
/intel/Linux/iostat/device/ALL/%util
```

A.3 The qsub VM Parameters

In the table below all VM related parameters for the qsub command line (wrapper) tool are listed and explained:

parameter	expected values	description
`img`	Path to image file or name of image	The image for the virtual guest, either a full path, relative path or name of the image that resides in the global image dir.
`distro`	one of the supported ones	Supported Linux OS distributions are: `redhat, debian, osv`
`ram`	RAM per VM in MB	If there is no value provided the possible max is chosen. That strongly depends on the amount of RAM dedicated to the host OS. In case the user requests more than one VM per node, the value is applied to each VM. So, the total amount required for all the VMs is `vms_per_node * ram`
`vcpus`	Number of virtual cpus per VM	If there is no value provided the possible max is chosen. That strongly depends on the amount of cores dedicated to the host OS. In case the user requests more than one VM per node, the value is applied to each VM. So, the total amount required for all the VMs is `vms_per_node * vcpus`
`vms_per_node`	Number of VMs per node	Amount of VMs per node, default is 1. Do not change it unless you have good reasons to do so, e.g. when allocating VMs on separate NUMA nodes or separating I/O intensive part of the application from the CPU intensive parts.
`metadata`	Metadata yaml file	Yaml-file with cloud-init targets for virtual guest customization. Note, there's only a selection of all targets available to the user, as this may have a major security impact.

`disk`	Persistent disk	Optional, persistent user disk. Will be mounted on the first node only of a job's resource allocation.
`arch`	CPU architecture	The CPU architecture for the virtual guest. For supported values please consult the qemu docs [44, 45]
`hypervisor`	hypervisor for the guest	Only KVM and sKVM are supported. Default is KVM.
`vcpu_pinning`	CPU pinning enabled	Indicates whether the pinning of virtual CPUs to physical CPUs is requested. Default is enabled.
`vm_prologue`	prologue script for the VM	Prologue script that is run under the user's account inside the virtual guest. The counterpart to the '-l prologue' resource request for the bare metal hosts.
`vm_epilogue`	epilogue script for the VM	Epilogue script that is run under the user's account inside the virtual guest. The counterpart to the '-l prologue' resource request for the bare metal hosts.
`vrdma`	vRDMA enabled	Indicates whether vRDMA is requested. Default is enabled.
`iocm`	IOcm enabled	Indicates whether the IO core manager is requested. Default is enabled.
`iocm_min`	min io cores	Min count of dedicated IOcm cores the user wants to utilize for his job, if IOcm is not disabled
`iocm_max`	max io cores	Max count of dedicated IOcm cores the user wants to utilize for his job, if IOcm is not disabled.
`fs_type`	file-system type	User can choose between shared file-system and a RAM disk, recommended default is the shared file-system
`-gf`	generate files	Not a '-vm key=value' pair, but a separate parameter that does not submit the generated artefacts, but prints the submission command and the filenames.

Example for job submission command line call

```
1) qsub -vm jobScript
```

Most simple variant to submit a virtual batch job. In this case default values for the PBS resources are applied by Torque and all default vm values are set by the qsub wrapper.

```
2) qsub -l nodes=2:ppn=16,walltime=01:00:00 \
      -vm img=/images/debian.img,distro=debian, \
          vms_per_node=2,vcpus=4,ram=4096,vcpu_pinning=true, \
          iocm=true,io_min_cores=2,io_max_cores=4\
          vrdma=true \
      jobScript.sh
```

This submit cmd explicitly requests 2 nodes with 16 cores and a walltime of 1h from Torque, as well as from qsub wrapper a debian image for the virtual guests. On each node 2 VMs should be started with 4 virtual cores, pinned to physical ones, and 4096MB RAM each, with vRDMA enabled, plus 2-4 dedicated cores for IOcm.

A.4 Scripts for VM-based Job Execution

A.4.1. Common Files

`config.sh`

Global configuration file that contains constants, paths, several settings to control behaviour and available features as well as default values for VM requests.

`functions.sh`

Contains all functions that are used by more than one user level script, i.e. the logging functionality

`root-config.sh`

Overrides selected variables that are defined in the global config `config.sh`, which is loaded automatically. The variables defined are required, by the scripts run as root user ,to be set to another value to work properly.

`root-functions.sh`

Beside functionality that is dedicated to the script run as root user, it overrides selected functions defined in the file `functions.sh`. It is loaded automatically, and is required by the scripts running as root user - e.g. logging function that writes to the syslog as well.

A.4.2. Job submission wrapper cmd line tool

`qsub`

Wrapper script that catches user's job submission cmd. It processes and clears the provided '`-vm <vm_resource_requests>`' parameters, generates (based on these parameters and defined default values) the **VM mgmt scripts** as well as the **job wrapper**. Then it passes the job submission onto Torque's `qsub` cmd line tool

A.4.3. Node Scripts

prologue

This script is run only by the first node of a job resource allocation. It prepares the shared file-system dir for the job (ensures it exists) or mounts a RAM disk instead if requested

prologue.parallel

Run by all nodes of a job resource allocation except for the first one. It mounts a RAM disk on each node if the job doesn't make use of a shared file-system.

epilogue

This script is run only by the first node of a job resource allocation. It unmounts the RAM disk if there is one for the job in place.

epilogue.parallel

Run by all nodes of a job resource allocation except for the first one. It unmounts the RAM disk on each node if the job doesn't make use of a shared file-system.

epilogue.precancel

Run by the first node of a job resource allocation, if the job is being canceled. It cleans up all files and directories the job has created.

A.4.4. VM Management Scripts

vmPrologue.sh

Executed only on the first node of a job's resource allocation. Usually referred to as rank0 in the parallel programming context. This script generates the individual parameter sets for each virtual guest. These parameter sets are used to create the domain.xml and other VM related files. Furthermore, it copies the required image file to a shared file-system or ram-disk (whatever is requested by the user), before spreading out to all nodes (including localhost) to run the `vmPrologue.parallel.sh` script. Depending on the global configuration this happens asynchronously or in parallel. As soon as the remote processes are started, the `vmPrologue.sh` script waits until all are complete.

In addition to that, it starts the user prologue on the bare metal host and the user vm prologue inside the first virtual guest of a job's resource allocation.

vmPrologue.parallel.sh

Executed on ALL nodes, boots the virtual guests on the local node.

vmEpilogue.sh

Executed only on rank0. Orchestrates the shut down of all virtual guests by executing the `vmEpilogue.parallel.sh` script on all nodes (including localhost). As soon as the remote processes are started, the `vmEpilogue.sh` script waits until all are complete.

In addition to that, it starts the `user epilogue` on the bare metal host and the `user vm epilogue` inside the first virtual guest of a job's resource allocation.

vmEpilogue.parallel.sh
Executed on ALL nodes, shuts down all virtual guests on the local node.

A.4.5. Root VM Scripts

vm prologue
Run on the first VM of a job's resource allocation, only. Will be executed after the boot process is complete, but before the SSH server is accepting connections.

vm prologue.parallel
Runs on all but the first VM of a job's resource allocation. Will be executed after the boot process is complete, but before the SSH server is accepting connections.

vm epilogue
Run on the first VM only of a job's resource allocation. Will be executed during the virtual guest's shutdown process.

vm epilogue.parallel
Runs on all but the first VM of a job's resource allocation. Will be executed during the virtual guest's shutdown process.

vm epilogue.precancel
Run on the first VM only of a job's resource allocation. Will be executed when the job is canceled. May be utilized to trigger checkpointing of applications.

A.4.6. Job Execution Wrapper Script

jobWrapper.sh
Prepares the virtual job execution environment by setting the `PBS_<something>` variables. Further, it starts the actual job inside the virtual guest.

The **user scripts** are generated and submitted by the user

user jobScript
User's job script that is executed on bare metal hardware or in a virtual guest, depending on '`-vm`' parameters provided for the job. It is started on the first node only.

user prologue
Users can optionally submit a prologue script with their job, that is run on the first physical node under their user, before the job script starts.

user epilogue

Users can optionally submit an epilogue script with their job, that is run on the first physical node under their user, after the job script has finished.

`user vm prologue`
Counterpart to the `user prologue`, but for the virtual environment.

`user vm epilogue`
Counterpart to the `user epilogue`, but for the virtual environment.

A.5 Misc Files

A.5.1. VM related files

`domain.slg.xml` and **`domain.osv.xml`**

Operating system specific in terms of standard Linux guests (`.slg`) and OSv (`.osv`)

`domain-fragment-disk.xml` and **`domain-fragment-metadata.xml`**
These template files define the virtual guests hardware-environment, i.e. hard-disks, network-connectors, and more. File `domain.xml` contains placeholders for the disk and metadata XML fragments

`metadata.*`
Metadata yaml files that are operating system specific. They control the boot sequence of the virtual guest by defining mount points for the virtual guest's (file `/etc/fstab` for example), installing required packages (e.g. nfs, ssh) and enabling services and commands at the end of the boot process.

A.5.2. Host OS related files

`99-mikelangelo-hpc_stack.sh`
Environment profile. This one is mandatory, since it defines the base path of the wrapper scripts installation. Furthermore, it enables users to call the wrapper instead of Torque's qsub, by prepending the `$PATH` environment variable.

`module_file`
There is an optional module's file available for the modules system [46], commonly in place in HPC environments.

A.6 Configurable Parameters

The parameters in the table below are defined in the file `config.sh` and are intended to be set by the administrators globally.

Config Parameter	Description
DISABLE_MIKELANGELO_HPC STACK	Disables the qsub wrapper in terms of not parsing any parameters and passing on the call directly to Torque's qsub.
DISABLED_HOSTS_LIST	Regex for host list that is disabled for VM job execution.
PARALLEL	Execute boot processes on the remote nodes asynchronously or sequentially. Parallel execution is recommended, since huge jobs may otherwise hit the timeout for the prologue phase.
ALLOW_USER_IMAGES	Indicates whether users are allowed to submit images with their job.
IMAGE_POOL	Absolute path to directory on a shared file-system that contains images available to users.
HOST_OS_CORE_COUNT	Cores reserved for the physical host. Greater or equals 0.
HOST_OS_RAM_MB	RAM in MB reserved for the physical host. Greater than 0, should match host os requirements as this is used to calculate RAM available to guests.
MAX_VMS_PER_NODE	Maximum count of VMs per node that cannot be exceeded.
STATIC_IP_MAPPING	Use a static mapping of mac addresses to ip addresses, instead of a DHCP server. Recommended value is false.
TIMEOUT	Timeout in seconds for remote processes to complete booting or destruction of VMs. Must be lower than Torque's timeout for pro/epilogue.
SERVER_HOSTNAME	Short hostname of server.
REAL_QSUB_ON_SERVER	Path to qsub on the submission front-end, used by the qsub wrapper to call Torque's qsub.
REAL_QSUB_ON_NODES	Path to qsub on the compute nodes, used by the qsub wrapper to call Torque's qsub.
IOCM_ENABLED	Indicates whether IOCM is enabled at all. If set to false the default and user settings are ignored for iocm.
IOCM_MIN_CPUS	Min amount of cpus that are always reserved for IOCM

IOCM_MAX_CPUS	Max amount of cpus that are always reserved for IOCM
VRDMA_ENABLED	Indicates whether virtual RDMA is enabled. If set to false the default and user settings are ignored for vRDMA.
VRDMA_NODES	Regex for list of nodes that are equipped with required hardware.
SNAP_MONITORING_ENABLED	Indicates whether monitoring with snap is enabled.

A.7 Default Values for VM-based Jobs

The default values for all mandatory parameters, listed below, are defined in the global configuration file `config.sh`.

Config Parameter	Description
FILESYSTEM_TYPE_DEFAULT	Defines where to place the images for virtual guests. Shared file system '$FILESYSTEM_TYPE_SFS' and RAM disk '$FILESYSTEM_TYPE_RD' are accepted
IMG_DEFAULT	Default image for virtual guests.
DISTRO_DEFAULT	Distro of the guest's image, depends on the default image. Supported OS (families) are 'debian', 'redhat' and 'osv'
ARCH_DEFAULT	CPU architecture of the default image. Usually 'x86_64'
VCPU_PINNING_DEFAULT	Recommendation is to enable it. Boolean value (true/false) expected.
VCPUS_DEFAULT	Default amount of virtual CPUs per guest
RAM_DEFAULT	Default amount of RAM dedicated to each virtual guest.
VMS_PER_NODE_DEFAULT	Recommended is one. Must be greater or equal to 1.
DISK_DEFAULT	Persistent user disk, mounted in the first VM of a job's resource allocation. Recommended is none (=empty).
HYPERVISOR_DEFAULT	Accepted values are 'kvm' and 'skvm'.
VRDMA_ENABLED_DEFAULT	Recommendation is to enable it. Boolean value (true/false) expected. Will be ignored if

	VRDMA_ENABLED is set to false.
IOCM_ENABLED_DEFAULT	Recommendation is to enable it. Boolean value (true/false) expected. Will be ignored if IOCM_ENABLED is set to false.
IOCM_MIN_CPUS_DEFAULT	Recommended value is '1'. Must be greater or equals 1.
IOCM_MAX_CPUS_DEFAULT	Recommended value is '4'. Must be greater or equals IOCM_MIN_CPUS_DEFAULT.

A.8 Environment Variables

A.8.1. Global Environment Variables

The global environment variables are defined in file

```
/etc/profile.d/99-mikelangelo-hpc_stack.sh
```

Its content is listed in the table below.

Name	Description	Note
MIKELANGELO_BASE_DIR	The base directory of the wrapper framework.	Mandatory
SCRIPT_BASE_DIR	Contains all files referenced in 4.4.1.1	Do not edit
TEST_BASE_DIR	Contains several tests to validate functionality.	Not needed for production

A.8.2. User Environment Variables

All environment variables that can be set in the user space are listed in the table below.

Name	Description
DEBUG	Enables debug messages.
TRACE	Enables trace messages.
SHOW_LOG	Shows the debug log immediately after submission.
SHOW_PBS_LOG	Shows the server and scheduler log immediately after submission.

KEEP_VM_ALIVE	Keeps the VMs running at the end of the job script until a flag file comes into existence that triggers continuation, or the walltime is hit, besides cancellation of the job. Useful for debugging of virtual guests.
DISABLE_MIKELANGELO_HPCSTACK	Can be use to disable the framework for job execution in virtual environments. Commands will be directly passed on to Torque's qsub.